Ficha Técnica

Proyecto Fin de Carrera

Título: Adaptación de algoritmos de reconocimiento visual para la implementación de juegos para niños basados en dispositivos Tabletop.

Autor: Alejandro Enjuanes Traín

D.N.I.: 76922080Y

Titulación: Ingeniería Informática

Directores: Sandra Baldassarri Santalucía

Javier Marco Rubio

Departamento: Informática e Ingeniería de Sistemas

Centro: Centro Politécnico Superior

Universidad: Universidad de Zaragoza

Fecha: Diciembre 2010

Adaptación de algoritmos de reconocimiento visual para la implementación de juegos para niños basados en dispositivos Tabletop

Resumen

La idea principal de este proyecto fin de carrera (PFC) consiste en conseguir que los niños muy pequeños puedan interactuar con software y elementos informáticos de una manera más natural, sin depender de los dispositivos básicos, como ratón o teclado, sino usando cualquier tipo de juguete o con sus propias manos.

Para lograr este propósito, este PFC se centró en el estudio y ampliación de un software de reconocimiento visual para dispositivos *tabletop*. El trabajo se realiza sobre NIKvision, una mesa de interacción tangible especialmente diseñada para su uso por niños pequeños.

En primer lugar se ha realizado un estudio de los elementos del sistema de NIKvision para, posteriormente, realizar varias fases de análisis e implementación de las distintas funcionalidades necesarias.

La primera fase consiste en la búsqueda de reglas para poder crear marcadores personalizados, con la finalidad de que sean capaces de adaptarse a las distintas formas que poseen los juguetes y que puedan tener una determinada forma o significado, no sólo para el ordenador, si no también para los niños (formas de animales, objetos, etc.).

La siguiente fase consiste en la modificación del *framework* de ReacTIVision para permitir la captura de dibujos realizados por los niños en un folio de papel. Estos dibujos estarán marcados con un marcador personalizado diseñado en la fase anterior, para luego poder trabajar con ellos en juegos o aplicaciones que usen la mesa como soporte.

La última fase consiste en permitir la detección de movimientos y pulsaciones de las manos de los niños sobre la mesa, así como la detección de juguetes “planos” (juguetes que no lleven marcas especiales en su base) y la orientación que éstos tengan.

A partir de cada una de las mejoras se han creado varios juegos para comprobar su funcionamiento. También se ha creado un juego que recoge todo lo implementado en la fases anteriores. El juego captura uno o varios dibujos, realizados en un folio con un *fiducial* personalizado, para luego poder interaccionar con las versiones digitales de éstos. Los dibujos se muestran en la superficie del *tabletop* repetidas veces para que el niño pueda golpearlos con sus manos, haciendo que desaparezcan, o desplazarlos a través de la pantalla con sus manos.

Índice General

Capítulo 1. Introducción

En este primer capítulo se introduce el proyecto fin de carrera explicando las motivaciones iniciales, el contexto en el que se ha realizado, los objetivos que se persiguen, una breve descripción de las herramientas que se han usado y, en último lugar, la estructura que se sigue en la memoria.

1.1 Motivación inicial

Actualmente el acceso a sistemas informáticos se ha extendido considerablemente y ha llegado a una gran variedad de usuarios. No obstante existe una franja de edad que, hasta el momento, no se ha tenido muy en cuenta a la hora de desarrollar software: la de los niños pequeños, de edades comprendidas entre los 3 y 6 años.

El control de los dispositivos de interacción normales, tales como ratones, teclados o joystick, les resulta muy difícil a los niños pequeños, debido a que no tienen muy desarrolladas sus capacidades cognitivas y psicomotrices. Por estas dificultades el niño, normalmente, no se siente motivado a utilizar los sistemas informáticos. Por otra parte existen diversos estudios que dicen que en esta etapa de crecimiento el niño necesita relacionarse con otros niños de su misma edad y los dispositivos actuales no suelen dar posibilidades para una buena colaboración entre varios niños jugando de forma conjunta y cooperativa [ChiCI].

Una de las formas que se utilizan para que el niño se comunique de forma más fluida con los sistemas informáticos es utilizando interfaces tangibles. Una interfaz tangible es una forma de comunicar información digital con una computadora a través de objetos corrientes, usándolos de forma natural y por varias personas a la vez. En el caso del *tabletop*[1] donde se va a realizar el proyecto los objetos que se usan son juguetes, ya que les resultan agradables y están acostumbrados a usarlos, adquiriendo una predisposición a usar el sistema tangible, jugando con él, además de poder utilizar también sus propias manos. El uso de interfaces tangibles es bueno para los niños, ya que permite la manipulación física de objetos y la relación con otros niños, mejorando las capacidades antes citadas [HEBJ].

Una de las posibilidades es usar un tipo de interfaz tangible denominado *tabletops* en las que el niño interacciona con una mesa simulando el modo de juego que realizaría con juguetes normales. Los *tabletops* tienen desarrolladas técnicas *multitouch* para permitir la interacción con la mesa aunque, en la franja de edades que se analizan, estas técnicas son demasiado complejas para los niños (las formas de interaccionar con la mesa son poco intuitivas para ellos y necesitan de una motricidad más desarrollada). Es por eso que se procedió a usar *tabletops* tangibles que detectasen las acciones y movimientos, pero de manera más básica, centrándose en acciones más simples, como desplazamientos, golpes, es decir, motricidad gruesa.

Como consecuencia de lo citado anteriormente se ha buscado el desarrollo de un dispositivo con el que los niños puedan jugar, interaccionando con él de la misma forma que lo harían normalmente: usando juguetes, dibujos y/o con las manos. Al no existir

1 Las palabras y términos en Itálica poseen una breve descripción de su significado en el Anexo 1, Glosario.

ningún *framework* que permitiera este tipo de interacción, se procedió a analizar la posibilidad de adaptar ReacTIVision[2] a nuestras necesidades para realizar mejoras que permitiesen la interacción de los niños con el *tabletop* NIKvision. Estas mejoras incluyen: el uso de juguetes para interactuar, el uso del *tabletop* por varios niños al mismo tiempo y la poca necesidad de realizar un manejo de movimientos precisos.

1.2 Contexto de desarrollo

Este proyecto se ha realizado dentro del **Grupo de Informática Gráfica Avanzada (GIGA)**[3] del Departamento de Informática e Ingeniería de Sistemas de la Universidad de Zaragoza, en la línea de trabajo sobre las interfaces tangibles dentro de un proyecto denominado NIK*vision*[4].

NIKvision es un *tabletop* tangible que permite a los niños pequeños jugar con el ordenador manipulando juguetes, a los que están acostumbrados, sin necesidad de dispositivos específicos, como ratón o teclado. En la Figura 1.1 se observa la interacción de varios niños con una aplicación informática, a través de juguetes, en el *tabletop* de NIKVision.

Figura 1.1 Niños interactuando con el tabletop de NIKvision

Su funcionamiento está basado en la detección visual de objetos que se colocan sobre la mesa y se capturan a través de una cámara. La información obtenida de las imágenes capturadas se envía a un software de procesado que la trata para obtener nuevos datos que se necesiten para interactuar otras aplicaciones. Cuando toda la información de la imagen ha sido procesada, un proyector se encarga de representar la información gráfica del juego por debajo de la superficie de la mesa, para favorecer la interacción que se tenga que realizar en la aplicación.

2 Para más información sobre el funcionamiento de ReacTIVision ir al Anexo 2.
3 http://giga.cps.unizar.es/
4 http://webdiis.unizar.es/~jmarco/

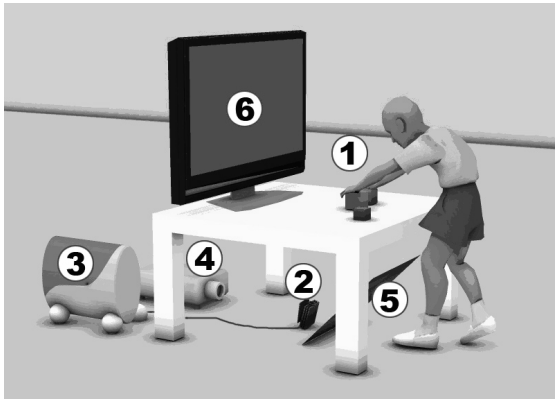

Figura 1.2 Diagrama de componentes del tabletop NIKvision

En la Figura 1.2 se observan las diferentes partes de las que se compone el *tabletop* de NIKvision (para otros *tabletops* pueden ser diferentes):

- Los juguetes que se utilizan para interactuar (**1**).
- Una cámara para leer los movimientos y posiciones de los objetos (**2**). Esta cámara puede ver en el espectro infrarrojo, apoyada por una dispositivo que emita luz de este mismo tipo. La razón por la que se utiliza este tipo de luz es la de conseguir una buena detección bajo cualquier tipo de iluminación exterior y que la luz de iluminación de la cámara no interfiera con la imagen proyectada sobre la superficie.
- Un ordenador que se encarga de procesar la información leída a través de la cámara y ejecutar el software del juego (**3**).
- Un proyector (**4**) que plasma una imagen sobre la superficie de la mesa para favorecer la interacción con el juego, ayudado por la reflexión sobre un espejo (**5**).
- Una pantalla donde se visualiza el juego (**6**).

Este PFC utiliza ReacTIVision, un *framework* de libre distribución para la interacción humana con el ordenador. ReacTIVision realiza un procesado de imágenes provenientes de una cámara para detectar objetos marcados con un identificador gráfico (denominado *fiducial)* y objetos de forma circular (denominados *fingers*). Después de realizar este procesado muestra por pantalla los datos obtenidos (tipo, identificador, posición, etc.) y los envía, mediante un protocolo de información denominado *TUIO*, a otras aplicaciones para que lo gestionen de la manera que lo necesiten. El tratamiento que ReacTIVision realiza sobre las imágenes utiliza la librería de tratamiento multimedia SDL. En los Anexos 2, 3 y 4 se encuentra detallada la forma de funcionamiento, características y funciones de ReacTIVision, el protocolo TUIO y la librería SDL, respectivamente.

1.3 Objetivos

El objetivo principal del PFC consiste en lograr que los niños de entre 3 y 6 años pueda interactuar de forma natural con un sistema informático, a través de juegos, en los que se utilizan diferentes tipos de juguetes, las manos, etc. Como, actualmente, no existe ninguna opción software en el mercado que realice este tipo de detección, se ha decidido ampliar las funcionalidades existentes dentro del *framework* de ReacTIVision para favorecer esa interacción.

Para llevar a cabo el desarrollo del objetivo principal es necesario definir una serie de objetivos secundarios que permitirán ampliar el *framework* de la forma necesaria:

- Personalización de identificadores gráficos para adaptarlos a las necesidades de juegos infantiles, basándose en la topología de los marcadores que usa ReacTIVision.

- Captura de dibujos en papel a través de la superficie de la mesa para poder utilizarlos en otras aplicaciones.

- Reconocimiento y detección de manos y objetos planos para que los niños sean capaces de interactuar con la mesa también con ellas, aportando también la orientación de éstos.

Para comprobar el correcto funcionamiento de las mejoras llevadas a cabo en este PFC se procederá a su incorporación en varios juegos interactivos que las utilicen en el entorno de NIKvision.

1.4 Herramientas

En este apartado mencionaremos las aplicaciones software y entornos de trabajo más importantes que se han usado para la realización del proyecto:

ReacTIVision [REAC]

Plataforma de visión por computador de código libre creado para el seguimiento e identificación de identificadores gráficos (*fiduciales*) añadidos a objetos, además de permitir acciones *multitouch* mediante el uso de los dedos. Además ReacTIVision es multiplataforma lo que permite que los cambios realizados se puedan ejecutar en varios sistemas operativos. Más información sobre ReacTIVision y su funcionamiento en el Anexo 2.

Microsoft Visual Studio 2005. Service Pack 1.

Gestor de proyectos y entorno de trabajo de Microsoft[5] que permite crear proyectos para aplicaciones de consola y de ventana. Posee opciones de depurar el código mientras se está ejecutando para observar los valores que tienen las variables y zonas de memoria. También se probó usar otras versiones del mismo gestor, produciendo errores de compilación que en la versión usada no se daban.

Fidgen [FID]

Aplicación software que genera *fiduciales* para ReacTIVision a partir de algoritmos genéticos, basándose en ciertos parámetros, como el color de fondo (blanco o negro), el número de elementos en su interior, etc. Esta aplicación se ha usado para deducir las reglas de generación de *fiduciales* por ReacTIVision y para obtener la composición de los ficheros para así poder crear *fiduciales* personalizados y adaptados a las necesidades del momento.

5 http://www.microsoft.com/spanish/msdn/vstudio/express/vb/default.mspx

1.5 Organización memoria

La memoria se organiza de la siguiente forma:

- **Capítulo 1. Introducción**: En este capítulo se incluye la motivación inicial del proyecto, contexto en el que se encuentra, los objetivos que se deben cumplir las herramientas usadas y la organización de la memoria.
- **Capítulo 2. Personalización de *fiduciales***: En este capítulo se incluye la explicación y el análisis de cómo se pueden crear *fiduciales* para simplificarlos y adaptarlos a necesidades estéticas y de forma.
- **Capítulo 3. Tratamiento de dibujos**: En este capítulo se aborda el tema de la captura y del tratamiento de dibujos a través de la mesa de trabajo: análisis de requisitos, análisis del proceso de detección de *fiduciales*, proceso de captura de la imagen, tratamiento posterior del dibujo y configuración de los valores de tamaño.
- **Capítulo 4. Tratamiento de blobs**: En este capítulo se incluye la explicación del proceso de detección de las interacciones de las manos sobre la mesa, y de otros objetos planos con orientación: análisis de requisitos, proceso de detección, calculo de la orientación de los objetos, envío de la información y configuración de los valores de tamaño.
- **Capítulo 5. Resultados**: En este capítulo se incluyen los resultados de cada implementación, con imágenes de su funcionamiento y de su aplicación en diferentes juegos desarrollados para el *tabletop* NIKvision.
- **Capítulo 6. Conclusiones:** En este capítulo se incluye la opinión personal sobre el proyecto, valoración y experiencia conseguida, demostración del cumplimiento de los objetivos iniciales e ideas para establecer una continuación o mejora de lo implementado.

Además se incluyen los siguientes anexos que complementan el trabajo realizado:

- **Anexo 1. Glosario de términos:** En este anexo se incluye la definición de los términos más relevantes usados en la memoria.
- **Anexo 2. Estructura de ReacTIVision:** En este anexo se explica la estructura software del *framework* de ReacTIVision, su funcionamiento y su interfaz gráfico, explicando las modificaciones o mejoras que se han realizado con el proyecto en esta parte.
- **Anexo 3. Estructura TUIO:** En este anexo se explica el funcionamiento del protocolo de comunicación que se utiliza en el proyecto, explicando las modificaciones o mejoras que se han realizado con el proyecto en esta parte.
- **Anexo 4. Tratamiento de la librería SDL:** En este anexo se explica el funcionamiento de la librería SDL y la documentación de las funciones que se han usado.
- **Anexo 5. Desarrollo temporal:** En este anexo se indica el tiempo empleado en el desarrollo del proyecto en cada una de las fases.
- **Anexo 6. Otros Tabletops existentes:** En este anexo se comentan algunos *tabletops* existentes, las características que poseen y la comparación con el *tabletop* en el que se ha llevado a cabo el proyecto.

Capítulo 2. Personalización de fiduciales

Para que la interacción de los niños se realice de forma más similar a la que suelen utilizar jugando como lo hacen normalmente, es necesario que los objetos que usen sean juguetes. Para que la cámara pueda detectar el tipo de juguete que se está usando es necesario que, a los juguetes, se les incluyan unos identificadores gráficos o *fiduciales* en su base. Al poder ser los juguetes de cualquier forma y tamaño, los identificadores gráficos que el propio ReacTIVision proporciona, pueden adaptarse con facilidad a los juguetes, o no. Por esta razón se ha determinado la necesidad de nuevos formatos de *fiducial* para poder adaptarlos a la forma de los juguetes que se van a usar para interactuar.

En este capítulo se explica en qué consisten los fiduciales, cómo se representan, la forma en que son tratados por ReacTIVision y el proceso de personalización de éstos, además de incluir algunos *fiduciales* personalizados que se han realizado.

2.1 ¿Qué son los fiduciales?

Un fiducial es una imagen o marca usada para identificar un objeto mediante un sistema de detección visual. Esta imagen puede dar información de tipo muy variado, como la identificación del objeto que es, su orientación y la posición de éste. El diseño de los *fiduciales* suele ser sencillo, permitiendo que mínimas variaciones en la imagen puedan dar lugar a un gran número de estos identificadores. Este diseño también suele ser simplista para que la CPU requiera de pocos recursos para poder identificarlos. En la Figura 2.1 se puede observar dos juguetes y como se le adaptarían los *fiduciales*.

Figura 2.1 Juguetes y adaptación a ellos de los *fiduciales* de ReacTIVision

Las imágenes que se utilizan como *fiduciales* consisten en la sucesión de dos colores, blanco y negro, ya que los sistemas de detección visual suelen emplear el espectro infrarrojo para discriminar lo que es la marca del resto del objeto.

Los *fiduciales* que ReacTIVision proporciona e interpreta tienen forma de ameba, ya que han sido diseñados para optimizar la distancia entre los contornos negros y blancos para que la cámara los detecte bien, usando círculos, que pueden ser concéntricos. Estos círculos proporcionan una orientación que puede ser usada en el procesado de la información. La orientación se deduce porque los círculos negros siempre se colocan en

el lateral opuesto a los círculos blancos, si esto no sucede así, el *fiducial* se sigue considerando válido, pero carecerá de orientación [CH09]. En la Figura 2.2 se observa la forma de un *fiducial* de ReacTIVision.

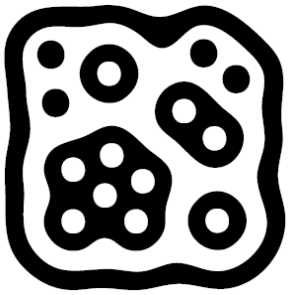

Figura 2.2 *Fiducial* usado por ReacTIVision

La forma de ameba de los *fiduciales* de ReacTIVision es siempre cuadrada, lo que supone una limitación según el juguete que se vaya a usar, pudiendo suceder que no se adapten bien a su forma cuadrada. A esto se le añade que en ocasiones es posible que se necesite que los fiduciales tengan un significado gráfico. Debido a estas razones se considera necesario poder personalizar los *fiduciales* dependiendo del juguete al que se le quiera aplicar o si se le quiere dar un significado mediante el dibujo.

2.2 Representación de los fiduciales

La información de cada *fiducial* viene dada por un árbol que se genera a partir de las zonas en blanco y negro que hay en él. La construcción del árbol se realiza tomando como nodo raíz el color más externo del fiducial, después se saca el número de zonas de color opuesto que hay dentro de la zona inicial. Para cada nueva zona detectada se vuelve a realizar el paso anterior, y así hasta que todas las zonas del fiducial se han analizado. En la Figura 2.3 se observa un *fiducial* y la representación correspondiente en forma de árbol.

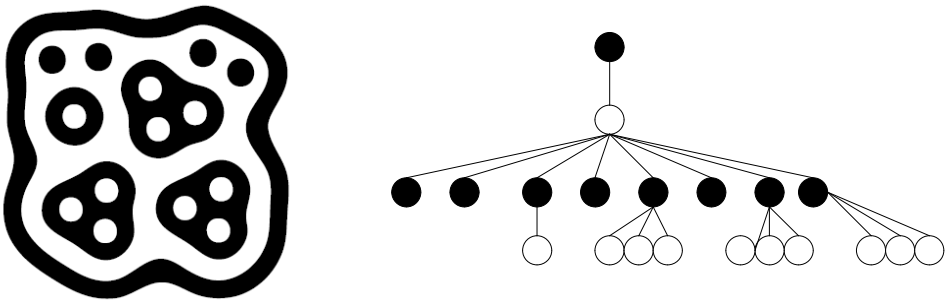

Figura 2.3 *Fiducial* y su árbol correspondiente

Dependiendo de la forma de tratarlo, los árboles generados pueden contener toda la información completa o sólo una parte. El proceso de minimización de los árboles consiste en eliminar las partes comunes y las partes que no dan ninguna información nueva, dejando únicamente las partes que se difieren. La representación óptima se realiza con 3 o más niveles de profundidad para tener suficiente robustez, ya que, usando menos, se pueden producir falsas identificaciones en la imagen captada, como considerarlo un *fiducial* distinto o no detectarlo.

Analizando el ejemplo anterior de la Figura 2.3 se puede eliminar el nodo raíz de color negro ya que la parte importante se encuentra en los nodos inferiores, debido a que el contorno del *fiducial* no aportará ninguna información sobre la diferenciación de éste con respecto a otro. Las zonas interiores son las que pueden variar y, por lo tanto, dar esa distinción. En la Figura 2.4 se observa como queda el árbol después de aplicarle una simplificación.

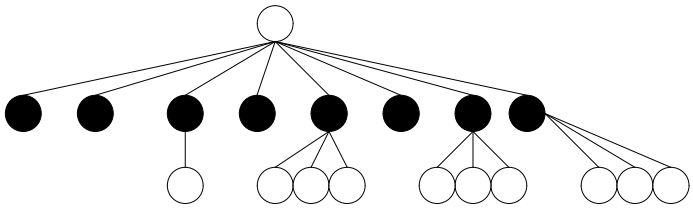

Figura 2.4 Árbol reducido del *fiducial* de la Figura 2.3

La detección de *fiduciales* no considera la forma geométrica de éste, sino el árbol que se obtiene aplicándole las reglas anteriores. La forma sólo influye para calcular la orientación, que puede ser o no ser necesaria dependiendo del uso que se le vaya a dar al *fiducial*. Este cálculo se realiza a partir de los círculos de menor tamaño del *fiducial* tal como se describe a continuación: se calcula el centro de la unión de todos los círculos del mismo color, obteniendo dos centros, uno para los blancos y otro para los negros. Cuando se han obtenido ambos centros se calcula el vector orientación que va desde el centro blanco hacia el centro negro, tomando como 0º el vector apuntando hacia arriba y aumentando con el giro en el sentido de las agujas del reloj. En la Figura 2.5 se puede observar el cálculo gráfico de la orientación.

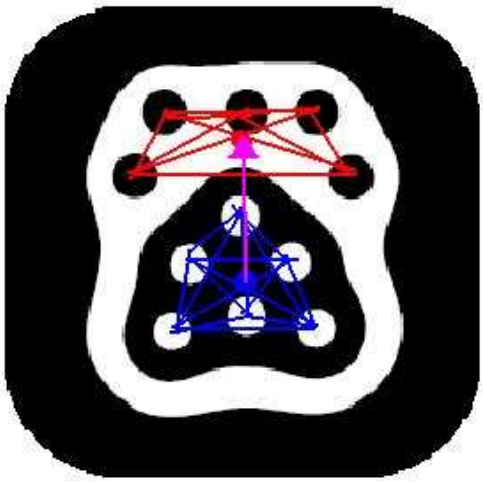

Figura 2.5 Cálculo gráfico de la orientación de un *fiducial* tipo ameba de ReacTIVision. La orientación viene representada por la flecha que va desde el centro que forman los puntos blancos, al centro que forman los puntos negros.

2.3 Formato del fichero

El sistema de detección de *fiduciales* que posee ReacTIVision obtiene los datos desde un fichero de texto ASCII, con extensión **.tree,** en el cual viene dada la información necesaria para distinguir cada uno de ellos. Estos ficheros son leídos de forma secuencial, asignándoles un identificador a cada uno de ellos, según el orden en el que se encuentren en el fichero.

En ReacTIVision las definiciones de los árboles se expresan de forma minimizada para reducir la información y de este modo poder almacenar mayor cantidad de *fiduciales*. El fichero está compuesto de tantas líneas de definición como *fiduciales* se quieran definir.

Cada línea tiene el siguiente formato:

– w / b: indica cual es el color del nodo raíz (sólo se necesita la información sobre el primer nodo, ya que los colores de cada nivel posterior son alternos).
– Ristra de valores compuesta por 0, 1 y 2. Cada número representa un elemento del árbol: 0, representa la raíz y sólo puede haber uno. 1, representa los nodos intermedios o ramas. 2, representa los nodos extremos u hojas. Esta ristra de valores se forma explorando el árbol el profundidad. Primero viene el 0, indicando el nodo raíz; luego un 1, seguido de varios 2, en función de cuantos nodos rama haya (si no hay nodos hoja no se pone nada); cuando un nodo rama se ha representado, se pasa al siguiente nodo rama, en caso de que lo hubiera.

Así, por ejemplo, la representación del árbol definido en la Figura 2.4 sería la siguiente: **w0122212221222121111.**

En el proceso de creación de fiduciales se ha de tener en cuenta que la representación de los árboles debe ser diferente para todos los *fiduciales*, ya que si dentro del fichero existen dos representaciones idénticas para 2 *fiduciales* distintos el sistema de detección visual considerará ambos como el mismo, asociando a los 2 el mismo identificador, que será el que venga definido en el fichero en primer lugar.

2.5 Personalización

Una vez analizada y comprendida la forma de expresar los *fiduciales* en forma de árbol y las reglas para representarlos en ficheros se puede explicar en detalle el proceso seguido para la personalización de *fiduciales*.

Debido a la existencia de juguetes con diferentes formas, los *fiduciales* proporcionados por ReacTIVision no se adaptaban bien, por lo que se procedió a crear algunos personalizados poder para adecuarlos a la forma del juguete. A continuación se explican varios intentos de personalización para los juguetes.

A partir de un juego de creación de música, que genera sonidos de batería mediante la colocación de marcas circulares (*fingers*) en zonas determinadas del *tabletop*, se desarrolló un regulador de velocidad. En la superficie del *tabletop* se dibuja una matriz de círculos que representan diferentes sonidos de la batería, teniendo que colocar una marca en ellos para poder generar sonido. El sonido se genera comprobando por columnas si los círculos tienen marca o no, esperando un tiempo hasta pasar a la siguiente columna. El regulador de velocidad permite elegir la velocidad con la que pasará a analizar la siguiente columna, es decir, el tiempo entre sonidos generados. El regulador tiene forma rectangular y funciona calculando la distancia del indicador, que se mueve simulando la regulación, al centro de un *fiducial* que se encuentra en el extremos superior. En las Figuras 2.6 y 2.7 se observan el formato del juego y la forma del regulador.

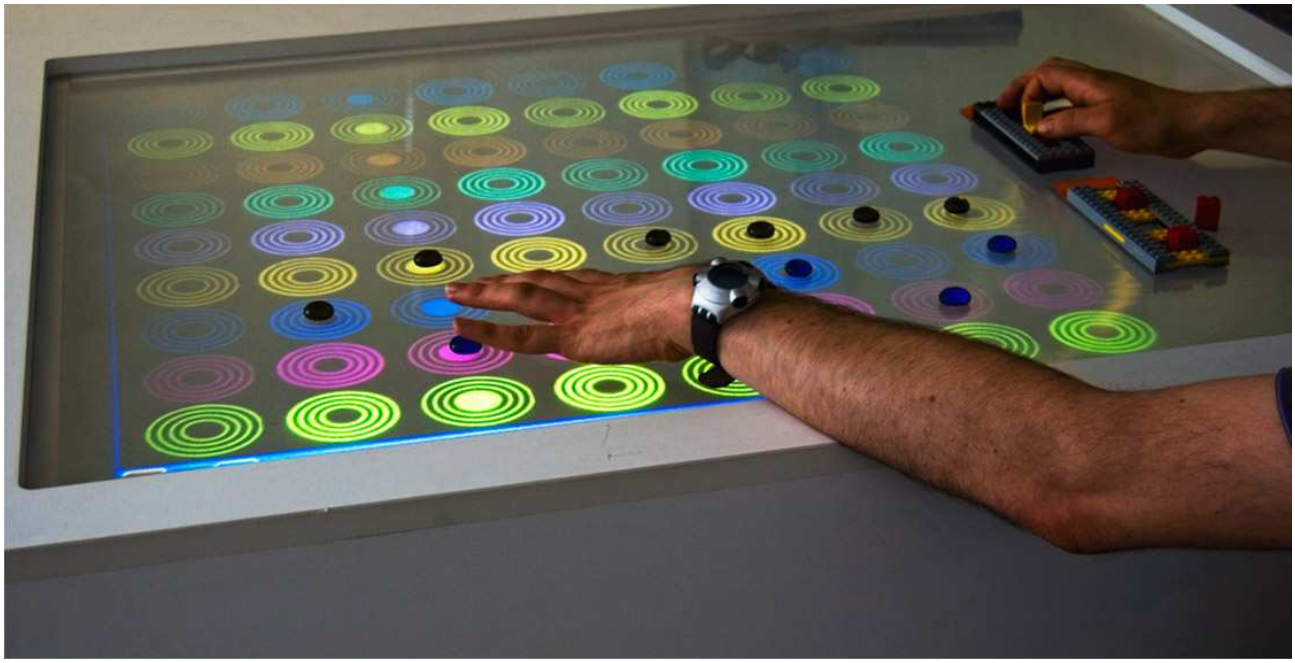

Figura 2.6 Juego de creación de música. En la superficie del *tabletop* se muestra la matriz de círculos en los cuales se ponen las marcas para generar los sonidos.

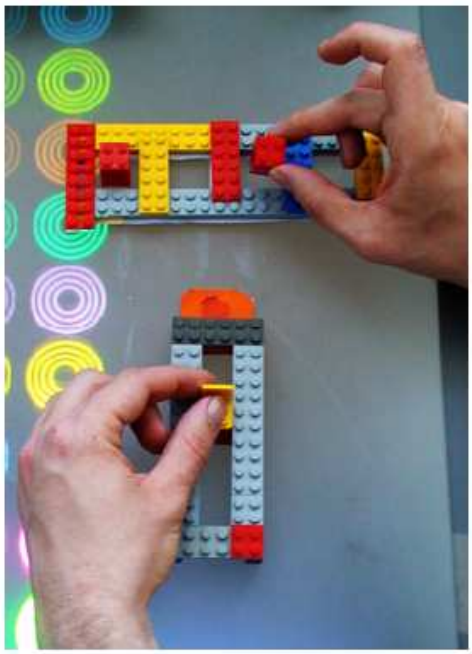

Figura 2.7 Dispositivo regulador del juego de música. La parte naranja corresponde a la zona dónde está colocado el *fiducial* de ReacTIVision. La parte amarilla es el regulador con el que se calcula la velocidad de generación de sonido.

Para evitar colocar el *fiducial* en el extremo superior y que quedase parcialmente fuera de la estructura, como se ve en la Figura 2.7, se decidió crear un *fiducial* personalizado de forma alargada para que se pudiese adaptar a la parte inferior de los laterales del regulador y, de ese modo, que la velocidad se calculase respecto a la posición del *finger* a los centros de los laterales. En la Figura 2.8 se muestra la parte inferior del regulador y el *fiducial* que se diseñó para acoplarlo en su base.

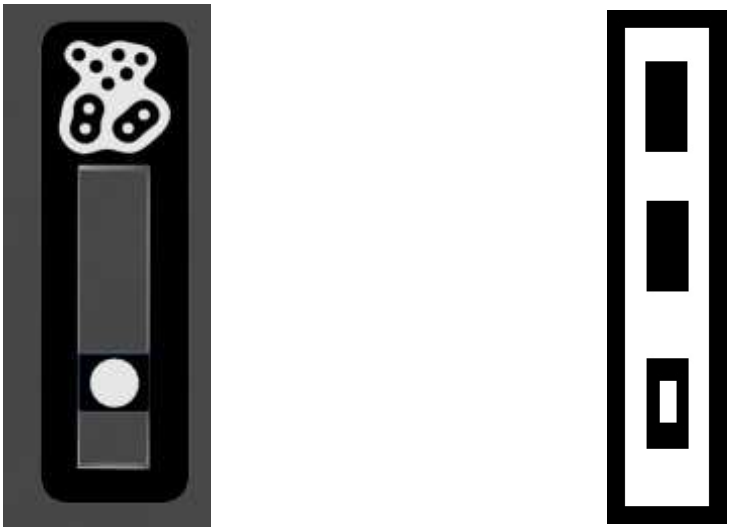

Figura 2.8 Base del regulador (izquierda) con *fiducial* en su extremo superior y *fiducial* personalizado (derecha) creado para acoplarlo a la base de los laterales y eliminar el *fiducial* superior.

Después de calcular su árbol y su representación en el fichero de reconocimiento (**w01112**) se procedió a probar si el sistema lo reconocía con claridad, observando que se producían errores en reconocimiento. Esto se debe a que el proceso de búsqueda de *fiduciales* en la imagen se realiza buscando formas cuadradas que contengan los niveles indicados en el árbol. Al ser el fiducial alargado, la superficie cuadrada que se buscaba era demasiado grande, por lo que, al ser menor la distancia de separación entre los 2 *fiduciales* de los laterales que la zona que se intenta buscar, no se reconocía ninguno de ellos. Se probó entonces a colocar el *fiducial* sólo en un lateral, pero el problema de detección persistía, ya que esta vez el *finger* se introducía en la región de búsqueda.

Al ser la región de búsqueda para el *fiducial* demasiado grande, la presencia del *fiducial* o del *finger* afectan en la detección por lo que el árbol generado es diferente al que *framework* tiene almacenado para detectar. Al comprobar que esta solución de mejora para la herramienta no era factible se procedió a mantener el dispositivo regulador de la forma inicial, llegando a la conclusión de que los *fiduciales* creados debían de poseer una forma mínimamente cuadrada.

A partir de la información conseguida sobre la forma que debían de seguir los *fiduciales* se estableció la creación de éstos con forma cuadrada. Por ello se procedió a la creación de un juego de naves espaciales en el cual las naves se representaba mediante un juguete para dirigirlas, con un botón para disparar. A ese juguete se le añadió en su base un *fiducial* personalizado para detectar su posición en la superficie del *tabletop.* En las Figura 2.9 se observan la forma del juguete usado y el *fiducial* que se le aplicó a su base.

A pesar de que el *fiducial* creado no es cuadrado el reconocimiento se realiza correctamente, debido a que en la región cuadrada de búsqueda no hay ningún elemento que pueda alterar el árbol. En la parte del diseño del *fiducial* se optó por simplificarlo mucho ya que al utilizarse sólo 2 *fiduciales*, uno para cada nave, no era necesario incluir muchas variaciones para detectar más *fiduciales*, ya que no los iba a haber. Complicando la zona interior del *fiducial*, y con ello el árbol que se genera, lo único que se consigue es aumentar la cantidad de elementos diferentes que se pueden detectar.

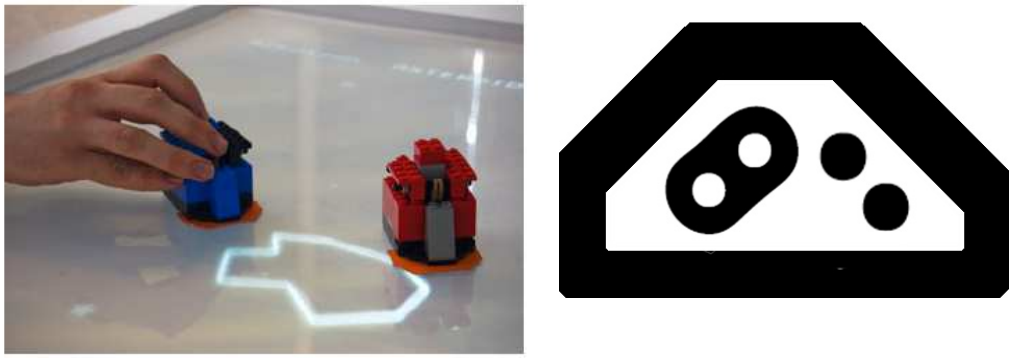

Figura 2.9 Juguete usado por el juego (izquierda) y *fiducial* personalizado aplicado a la base (derecha)

Con una simplificación excesiva puede ocurrir que se lleguen a detectar *fiduciales* falsos en otras formas que no lo sean, sin embargo, añadiendo demasiados elementos sucede lo contrario: al incluir demasiados elementos en un mismo espacio, el tamaño de estos tiene que ser disminuido, pudiendo aparecer problemas en la detección.

El *fiducial* debe ser lo suficientemente grande para detectar bien todos los elementos que lo componen, o bien, que la cámara tenga un resolución alta para llegar a captarlos todos. El hecho de simplificar el *fiducial* supone aumentar la robustez en la detección, ya que los elementos a analizar serán menores permitiendo que del *fiducial* sea reconocido independientemente del tamaño, evitando aumentar el tamaño de éste o del juguete.

Una vez que se consiguió reconocer *fiduciales* personalizados se procedió a dotar a éstos de una forma que supusiese una identificación visual de su significado al niño. A partir de cualquier dibujo en blanco y negro y se puede crear un dibujo con significado que, a su vez, venga representado por un árbol topológico. En la Figura 2.10 se puede observar esto mismo: el dibujo representa un lápiz, proporcionando al niño una información de que debe realizar algo gráfico con ello, pero además proporciona a ReacTIVision una información de *fiducial*, concrétamente **w012221**, del que además se puede extraer la orientación, convirtiéndolo en un *fiducial* válido. Éste *fiducial* se usa concretamente colocándolo en un papel para que el niño dibujo sobre él y poder capturar el dibujo realizado a través de la superficie del *tabletop*, como se explica en detalle en el siguiente capítulo.

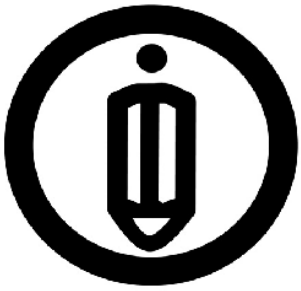

Figura 2.10 *Fiducial* válido con representación gráfica. La orientación saldría del vector que va desde el centro de las partes blancas del lápiz al punto negro que está encima.

Capítulo 3. Tratamiento de dibujos

Habitualmente los niños pueden jugar no sólo usando juguetes, sino también dibujando. Dado que el *tabletop* donde se trabaja posee una cámara que detecta lo que se coloca en la superficie, se consideró la opción de poder capturar dibujos de los niños mediante el *tabletop*. El niño dibujará sobre un papel que posee un *fiducial* determinado y al colocarlo sobre la superficie del *tabletop* será capturado para luego ser usado en otras aplicaciones.

Esta idea surgió a partir del trabajo de Zach Lieberman [ZL] que permite la captura de dibujos realizados sobre la propia mesa a través de un botón para luego interactuar con los elementos de la imagen (ver Anexo 6 para más información sobre diferentes *tabletops)*. La idea pareció interesante para aplicarla a los propios niños pero en vez de tener que dibujar sobre la mesa, hacerlo sobre el papel, forma que les resulta más cotidiana. Además permite que el niño no monopolice la mesa, permitiendo que los niños dibujen en sitios diferentes y luego carguen sus dibujos todos juntos. Para añadirle más sencillez al proceso de captura no hay que pulsar ningún botón como en la mesa de Lieberman, sino que se realizará automáticamente cuando se deposite el folio sobre la superficie.

En este capítulo se trata el análisis del funcionamiento de la detección de fiduciales y la modificación de la librería existente para poder realizar una captura de pantalla y poder realizar el procesado de ésta y consta de los siguientes apartados: análisis de requisitos del sistema, análisis del proceso de detección de *fiduciales*, proceso de captura de la imagen, tratamiento posterior del dibujo capturado y la configuración de los valores de tamaño del papel de dibujo.

3.1 Análisis de requisitos

En este apartado se incluyen los requisitos funcionales y no funcionales que deberá cumplir la implementación diseñada:

- RF-1: El *tabletop* deberá capturar dibujos que le sean puestos sobre la pantalla.
- RF-2: El *framework* de ReacTIVision debe poder parametrizarse decidiendo el tamaño y forma del papel de dibujo .
- RF-3: El dibujo capturado debe ser enviado a otra aplicación juego para que ésta lo utilice.
- RNF-1: El dibujo capturado por el *tabletop* debe de estar lo más limpio posible de zonas oscuras que se obtienen por el proceso de captura.
- RNF-2: La imagen debe ser orientada para quedar totalmente horizontal sin depender de la posición en la que el niño la haya colocado.
- RNF-3: La imagen debe se ser tomada cuando el folio lleve un tiempo suficiente sin moverse.

3.2 Análisis del proceso de detección de fiduciales

Durante el funcionamiento del *framework* de ReacTIVision la cámara se encarga de enviar *frames* de lo que está viendo al sistema para que éste los procese. Cada *frame* que la cámara envía al sistema se analiza para buscar algún *fiducial* con información para detectar. Para tratar los *frames* que la cámara está enviando continuamente se opera mediante estructuras del tipo Simple DirectMedia Layer (ver Anexo 4), almacenando cada *frame* captado por la cámara en un estructura para tratarla. Cada vez que en un *frame* se encuentra un *fiducial* que está definido en el *framework*, se muestra la información del identificador por consola. En la Figura 3.1 se muestra una captura de pantalla del sistema de detección en el que se ven 2 *fiduciales* que son reconocidos por éste, mostrando la información de su identificador en el centro de cada uno de ellos.

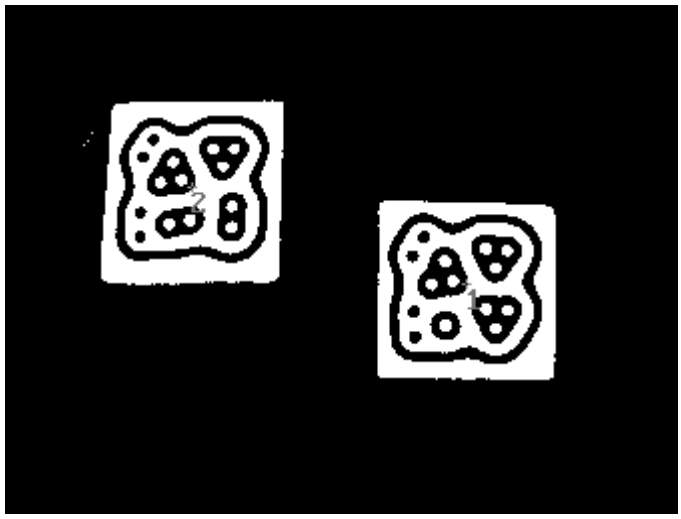

Figura 3.1 Pantalla de detección en información de fiduciales. Al *fiducial* de la izquierda le corresponde el identificador 2, al de la derecha el 1.

Durante el proceso de detección se realiza también un proceso de comprobación para determinar si el *fiducial* se encontraba en el *tabletop* y si se había detectado con anterioridad. Cuando el *framework* ha localizado todos los *fiduciales* del *frame* envía la información de los datos correspondientes a cada *fiducial* mediante el protocolo TUIO (ver Anexo 3) a otras aplicaciones que necesiten esa información.

3.3 Proceso de captura de imagen

El proceso de captura de la imagen se realiza durante el momento en que se detectan los *fiduciales*, ya que es cuando se procede a tratar el *fiducial* que indica la existencia de un papel con un dibujo sobre el *tabletop.* Por otra parte como la luz con la que se ilumina la parte inferior del *tabletop* es infrarroja, y la cámara está preparada para ver capturar ese espectro únicamente, los dibujos siempre se capturarán en color negro, aunque estén pintados con otro color.

3.3.1 Detección del papel

Durante el proceso de captura se comprueba si alguno de los *fiduciales* detectados en el *frame* es el buscado, que indica que hay un papel con un dibujo colocado sobre la superficie del *tabletop*. Cuando el *fiducial* es detectado se procede a comprobar el tiempo de permanencia, para determinar cuanto tiempo permanece quieto el folio, sin moverse. De este modo se evita que, cuando el niño coloque el folio sobre el *tabletop,* se capture una imagen errónea por la existencia de movimiento y se asegura la correcta captura de la imagen.

Para verificar el tiempo de espera se comprueban las posiciones del *fiducial* durante los siguientes *frames*. Para ello se almacena la posición en el *frame* anterior, en coordenadas x e y, y se comprueba con la posición en el *frame* actual si el folio se ha desplazado o por el contrario se ha quedado quieto. Si la variación respecto de las posiciones es menor que un valor umbral dado ese *frame* contará como válido, si no, no se contabilizará y se esperará al siguiente *frame* para comprobar.

El tiempo mínimo de espera se ha establecido como el equivalente a 20 comprobaciones válidas, ya que menos número de comprobaciones apenas permite que el folio sea colocado correctamente; y por el otro lado, más ejecuciones genera un tiempo de espera demasiado alto para resultar aceptable por los usuarios. Una comprobación válida es aquella en que la posición del *fiducial* entre 2 *frames* consecutivos ha variado menos que el umbral indicado.

Núm. Comprobaciones	T. resp. mejor caso(s)	T. resp. peor caso(s)	T. Medio(s)
10	0,97	2,43	1,7
20	1,7	3,4	2,55
30	2,68	5,8	4,24
40	3,08	6,04	4,56
50	4,69	7,13	5,91

Tabla 3.1 Tabla con los tiempos de respuesta mejores y peores con diferente número de comprobaciones.

En la tabla 3.1 se indican los tiempos calculados para diferente número de comprobaciones válidas. El tiempo de respuesta en el mejor caso es el tiempo medio de los mejores tiempos obtenido de los resultados dejando el papel fijo y sin tocarlo. El tiempo de respuesta en el peor caso es el tiempo medio de los peores tiempos obtenidos con el papel moviéndose, es decir, con el niño tocándolo sin poder dejarlo fijo. Con estos datos se calcula el tiempo medio como la suma de ambos tiempos divido entre 2.

Una vez que que el folio se considera que está quieto, se almacena la última posición obtenida en el reconocimiento y se envía un mensaje TUIO a otras aplicaciones indicando que se ha detectado un papel y que la captura de la superficie de la mesa se ha realizado con éxito.

3.3.2 Rectificación de la imagen

Una vez obtenidos los datos de la posición del *fiducial* en la superficie de la mesa y su orientación se establece el tamaño concreto de la zona del dibujo que se desea capturar. Debido a que en el proceso de captura sólo se ha basado en la posición del *fiducial* y la quietud del folio, y no en su orientación, éste puede haber sido colocado con cualquier orientación, provocando que la zona de la imagen que se va a capturar no sea la que contenga el dibujo o que salga cortado. Para mejorar el proceso de captura y para permitir que el niño pueda colocar el folio en la posición que el quiera, sin obligarle a que deba posicionarlo de un forma determinada, ya que le puede resultar molesto y quitar fluidez al proceso de captura, se procede a rotar la imagen para que la zona que corresponde al folio quede horizontal.

La rotación de la imagen se realiza tomando todo el *frame* y aplicándole un algoritmo de giro, dado por la librería SDL_rotozoom.h[6]. Este algoritmo gira la imagen obtenida considerando el centro de coordenadas en el centro de ésta, tantos grados como se indican por la orientación del *fiducial*. Al aplicar este giro el *fiducial* queda con una orientación de 0º, apuntando su vector de orientación hacia arriba. En las Figuras 3.2 y 3.3 puede observar una imagen con el folio capturado en una posición no horizontal y otra imagen rectificada con el folio colocado en horizontal.

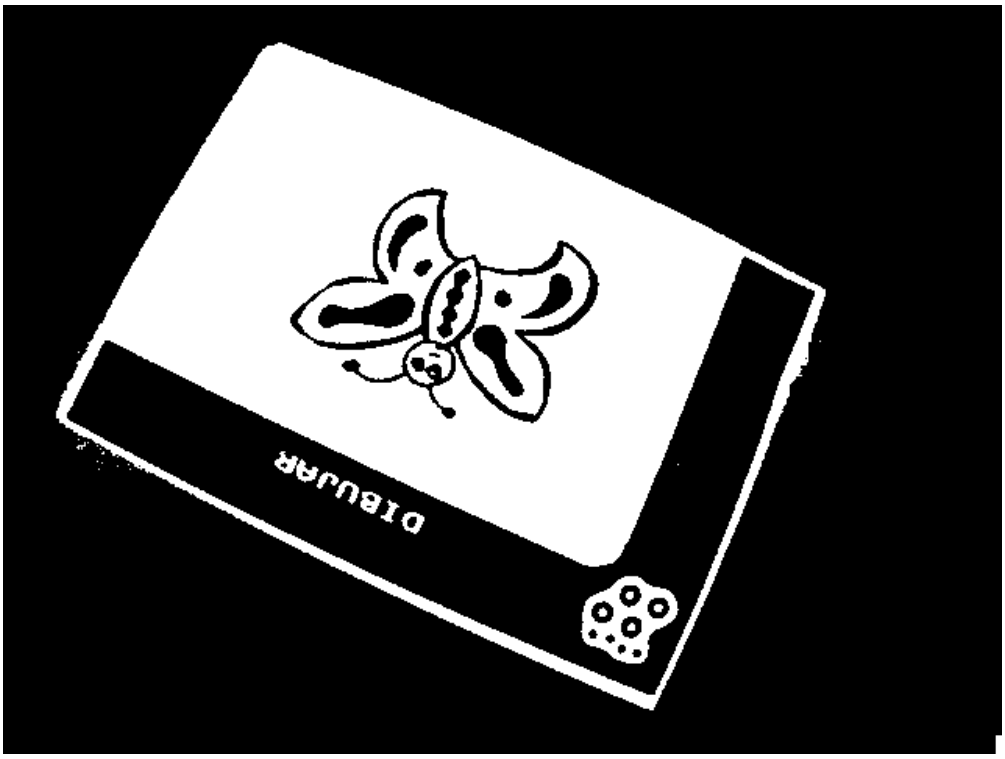

Figura 3.2 Imagen capturada mediante la superficie del *tabletop* antes de ser rectificada para que el folio quede horizontal. Si no se realizase el giro separar el dibujo del resto de la imagen sería más complicado.

6 Para más información ver Anexo 4.

Figura 3.3 Imagen rectificada de la Figura 3.2 en la que el folio se encuentra en posición horizontal.

Después de rotar el *frame* es necesario volver a calcular la posición del *fiducial*, ya que con el giro ésta habrá cambiado. El cálculo de la nueva posición se realiza usando algoritmos de transformación para imágenes, en este caso en concreto se utiliza el de rotación. A partir de la posición en la imagen sin rectificar y la orientación del *fiducial* se procede al cálculo de la nueva posición. El giro usado para la rectificación se realiza tomando como centro de coordenadas el centro de la imagen, por lo que se debe calcular la posición relativa a este centro, ya que la que el *framework* toma es con el centro en la esquina superior derecha. Para ello se toman la altura y anchura del *frame* y se procede al cambio de coordenadas mediante las siguientes fórmulas:

$$x' = x - anchura/2$$
$$y' = altura/2 - y$$

Una vez obtenida la posición relativa al nuevo centro se procede a calcular el punto en el que se encontrará ahora el *fiducial.* Para ello, usando como ángulo el que se ha obtenido para el giro, la orientación del *fiducial*, se aplican las siguientes fórmulas de transformación definidas en la Figura 3.4.

$$\begin{aligned} \mathbf{i'} &= \cos\alpha\,\mathbf{i} + \operatorname{sen}\alpha\,\mathbf{j} \\ \mathbf{j'} &= -\operatorname{sen}\alpha\,\mathbf{i} + \cos\alpha\,\mathbf{j} \end{aligned} \Rightarrow \begin{Bmatrix} \mathbf{i'} \\ \mathbf{j'} \end{Bmatrix} = \begin{pmatrix} \cos\alpha & \operatorname{sen}\alpha \\ -\operatorname{sen}\alpha & \cos\alpha \end{pmatrix} \begin{Bmatrix} \mathbf{i} \\ \mathbf{j} \end{Bmatrix}$$

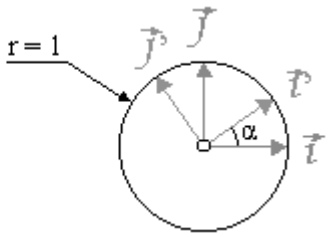

Figura 3.4 Aplicación de la matriz de rotación a un punto de un eje de coordenadas.

Una vez que se obtiene la nueva posición en la que se encuentra el *fiducial* después del giro, se vuelve a calcular la posición respecto al centro de coordenadas, esta vez establecido en la esquina superior izquierda.

Una vez obtenida la posición del *fiducial* se procede a seleccionar la región del *frame* que se necesita para obtener el dibujo. La imagen tratada hasta el momento contiene todo lo que la cámara ve, pero sólo se necesita la zona del folio que contiene el dibujo, desechando todo lo demás, por lo que se establece un recorte a la imagen para obtener esa región. La zona de recorte se establece a partir de la posición del *fiducial* tomando un anchura y una altura parametrizables mediante el *framework*. Hay que considerar también que la posición del *fiducial* da el centro de éste, por lo que hay que evitar que al recortar la imagen salga parte del *fiducial*. Para que no salga se establece que el punto origen del recorte no sea el centro exacto del *fiducial*, si no que esté alejado una distancia, también parametrizable. En la Figura 3.5 se observa la sección del *frame* que contiene el dibujo recortado.

Figura 3.5 Imagen recortada en la que se ha separado el dibujo del resto. Gracias a la altura y a la anchura establecidas se puede determinar el tamaño del dibujo. Usando una distancia a partir del centro del *fiducial* se ha conseguido que no salga ninguna parte de éste.

3.3.3 Limpieza de la imagen

Por mucho que se ajuste la posición de corte para evitar la captura de zonas que no tienen correspondencia con el dibujo, siempre quedan zonas negras, provocadas por las aberraciones en los extremos que posee la lente de la cámara que se usa. Cuando el folio se coloca en las zonas más alejadas del centro del *tabletop* sufre un abombamiento que hace que el folio no se vea rectangular, sino deformado. Como el recorte se realiza considerando una forma rectángular, si la imagen está deformada, surgirán manchas en negro correspondientes a la zona próxima al folio. Estas zonas en negro afectan al dibujo a capturar ya que, si no se eliminan, formarán parte de éste cuando se utilice en otras aplicaciones. Para eliminar estos errores en la captura se procede a realizar una limpieza de las zonas, que se sitúan siempre en los bordes de la imagen, mediante un algoritmo.

El algoritmo de limpieza realiza un borrado de todos los bordes (superior, inferior, lateral izquierdo y lateral derecho) realizando 2 pasadas a la imagen: en la primera se limpian los laterales y en la segunda las zonas superior e inferior.

Para realizar eficientemente las pasadas de borrado se trata la imagen como si fuera un matriz. En la primera pasada se recorre la matriz imagen por filas. El algoritmo se coloca en el primer píxel de la fila comprobando si éste es de color negro, si es así, le

cambia el color a blanco y avanza al siguiente para realizar la misma comprobación. Si el píxel que se encuentra es de color blanco, significa que ya se ha llegado a la zona del folio y se pasa al lateral opuesto para seguir con el borrado de esa parte. En el lateral opuesto el algoritmo se coloca en el último píxel de la fila y realiza las mismas acciones que en el otro lateral, pero en vez de hacerlo avanzando píxeles, lo hace retrocediendo. Cuando el algoritmo ha terminado de limpiar una fila, pasa a la siguiente, así hasta que se hayan recorrido todas las filas de la imagen. La segunda pasada realiza el mismo proceso que para los laterales, siendo en este caso el recorrido de la matriz por columnas, siendo el primer píxel que se evalúa el de la primera fila y los siguientes los de las filas sucesivas. Cuando se produce el salto al borde inferior del dibujo el píxel por el que se empieza a analizar es el de la última fila y los siguientes los de las filas anteriores.

Una vez que se ha realizado el proceso de limpieza de bordes se obtiene la imagen lista para usarla en otras aplicaciones, como se puede ver en la Figura 3.6.

Figura 3.6 Dibujo capturado con los bordes limpios de zonas negras por la aberración de la cámara.

La implementación que se ha dado al algoritmo de limpieza de bordes no tiene en cuenta los pequeños píxeles de suciedad que puedan surgir en el dibujo, ya que no influyen, siendo lo único que interesa eliminar las amplias zonas en negro que quedan en los bordes. En un primer momento se pensó en aplicar algoritmos específicos para eliminar las zonas del dibujo que estuviesen manchadas para que quedase más limpio, pero al tratarse de de dibujos realizados por los niños y capturados por una cámara es difícil discriminar lo que es parte del dibujo y lo que es suciedad. Un algoritmo de limpieza más complejo podría haber eliminado bastantes partes importantes del dibujo, por lo que se llegó a la conclusión de que lo mejor era realizar un borrado más sencillo y dejar esas manchas, antes que eliminar partes del dibujo.

3.4 Tratamiento posterior del dibujo

Durante el proceso de detección, en el momento en que se considera que el folio está quieto, se envía un mensaje informando de que se ha detectado el folio con el *fiducial*. El mensaje envía un identificador de *fiducial* falso, 500, ya que no interesa que lo use otra aplicación para nada más que indicar la presencia del folio. Cuando el juego recibe este mensaje sabe que tiene que cargar un archivo de dibujo en la dirección específica y se queda esperando a que se cree el archivo, es decir, hasta que el procesado de la imagen haya acabado. Ésto se realiza en una sección crítica en la que el

juego no continua ejecutándose hasta que ha leído una imagen, para evitar así que mientras el juego está leyendo una imagen, ReacTIVision la modifique.

Una vez que la imagen ha sido tratada y se da por buena es cuando pueden utilizarla otras aplicaciones. Después de que la imagen ha sido limpiada se procede al almacenamiento de ella en el ordenador, guardándola en formato *BMP.*

3.5 Configuración de los valores de tamaño

El folio que se usa para realizar y poder capturar el dibujo debe poder ser de cualquier tamaño, siendo el único requisito que posea el *fiducial* específico para detectarlo. Dejar que cualquier tipo de folio pueda ser usado para capturar los dibujos permite que los dibujos puedan ser de diferentes tamaños y adaptar a diferentes tamaños del *tabletop,* proporcionando una mayor flexibilidad pero impidiendo usar valores de corte fijos, ya que en algunos dibujos sobrarán partes y en otros faltarán elementos. Esto también sucede cuando se utiliza otra mesa de tamaño diferente, ya que la resolución de la imagen que la cámara obtiene es la misma, 640x480, quedando el dibujo más pequeño si la superficie del *tabletop* es mayor, o más grande, si su superficie es menor.

Por las razones expuestas anteriormente se ha establecido que los valores de altura y anchura de recorte, y el desplazamiento respecto al centro del *fiducial* sean configurables a partir del *framework*. La configuración y visualización de las opciones se desarrolló de forma similar al resto de configuraciones de valores en ReacTIVision. La activación del menú de configuración de tamaños se realiza cuando se pulsa la letra 'b' del teclado. Esto permite modificar el tamaño de la zona que se va a recortar por anchura (máximo 600 píxeles) y altura (máximo 400 píxeles). Los valores máximos se decidieron ya que la configuración de pantalla que se usa es 640x480. Para la configuración del desplazamiento se establece como máximo 50 píxeles de anchura, ya que proporcionalmente nunca ocupará más de ese tamaño en el folio. En la Figura 3.6 se puede observar el menú de configuración para los parámetros de configuración de altura, anchura y desplazamiento.

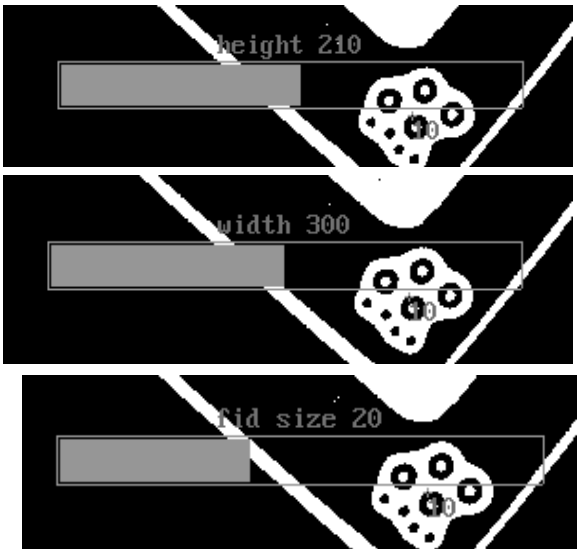

Figura 3.7 Parametrización de los valores de configuración: altura (arriba), anchura (centro) y desplazamiento (abajo).

Capítulo 4. Tratamiento de blobs

Los niños muy pequeños suelen utilizar sus manos para intentar interaccionar con las cosas de su alrededor, antes incluso que utilizando juguetes, para tocar, golpear, etc. Para favorecer que el niño pueda interaccionar sin necesidad de juguetes especiales, se requiere una funcionalidad para detectar los movimientos de las manos de los niños que se encuentren interaccionando con el *tabletop*. Por otra parte también era necesario permitir el reconocimiento de juguetes que no estuviesen marcados con un *fiducial.* Este tipo de juguetes son detectados por la forma de su base, como por ejemplo su área o geometría, siendo denominados juguetes “planos”. Además de detectar las manos y los juguetes planos, también se puede obtener la orientación de estos mediante los cálculos de los descriptores.

En este capítulo se describe el proceso de detección de manos y objetos planos (denominados *blobs*), el calculo de la orientación, el envío de la información y la parametrización de las opciones.

4.1 Análisis de requisitos

En este apartado se incluyen los requisitos funcionales que deberá cumplir la implementación diseñada:

- RF-1: El *framework* deberá ser capaz de capturar las manos y objetos planos.
- RF-2: El *framework* deberá ser capaz de calcular la orientación de los elementos anteriores.
- RF-3: Se podrá parametrizar el tamaño de los *blobs* en ReacTIVision.

4.2 Proceso de detección de blobs

El proceso de detección de *blobs* se realiza, al igual que la detección de *fiduciales*, en la zona de procesado de las imágenes (ver Anexo 2 para más información sobre las acciones que realiza ReacTIVision). El propio ReacTIVision realiza una detección de *blobs* específicos denominados *fingers,* que poseen forma circular y que pueden ser de varios tamaños, ya que la propia estructura del *framework* permite configurar el tamaño de detección. Cuando un *finger* es detectado por el *framework*, si se encuentra dentro del rango dado, se indica mediante una marca en la pantalla. En la Figura 4.1 se pueden observar varios *fingers* y los indicadores de detección.

.

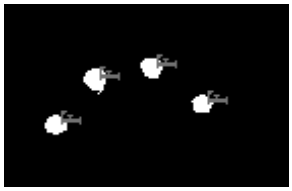

Figura 4.1 Varios f*ingers* detectados y sus indicadores.

Como el proceso de reconocimiento de los *blobs* comparte bastantes elementos con el que se realiza para los *fingers*, se realiza de la misma forma para ambos, si bien, en cada uno se realizan diferentes comprobaciones de tamaño y forma.

ReacTIVision realiza el proceso de detección de *blobs* durante el procesado de los *frames.* A cada *frame* que envía la cámara se le aplica una umbralización para dejar todos sus píxeles en blanco o en negro. Cuando se ha umbralizado el *frame*, se procede a segmentarlo en regiones de un mismo color (o blanco o negro), obteniendo varias regiones de diferentes tamaños y un centro para cada una de ellas. El centro de cada región se obtiene a partir de los píxeles más alejados de la región, en los laterales y los extremos superior e inferior, formando un rectángulo con ellos, siendo el centro el punto medio del rectángulo. Una vez que se tienen todas las regiones detectadas, se buscan aquellas que se encuentran dentro de un rango de tamaño, mínimo y máximo, discriminando el resto de zonas que se encuentran fuera de ese rango. Se ha modificado el proceso de búsqueda original para que a parte de comprobar el tamaño de los *fingers* también compruebe el tamaño de los *blobs* que no sean *fingers*, puesto que no tienen porqué ser iguales los 2 rangos. En la Figura 4.2 se puede observar las regiones detectadas y el rectángulo que las rodea.

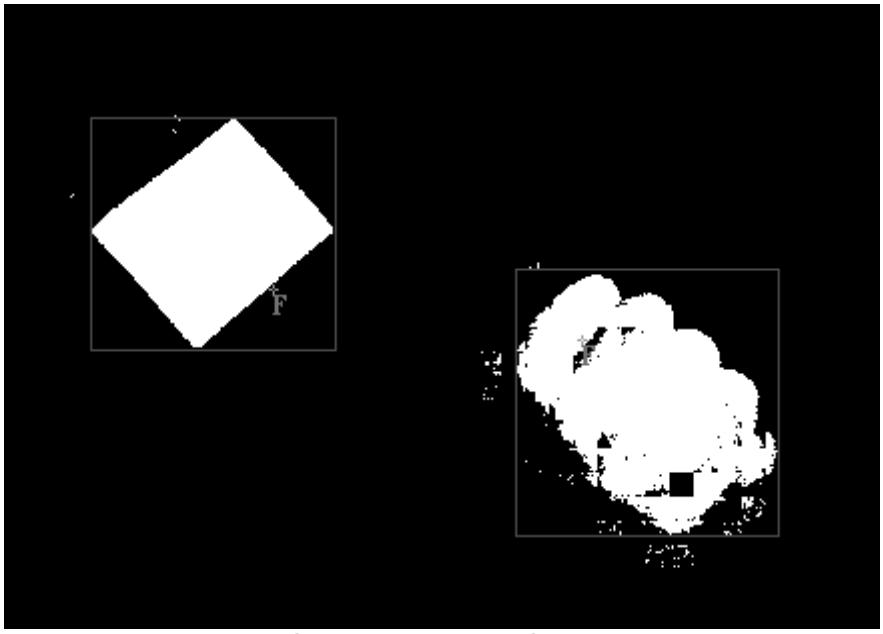

Figura 4.2 Detección de zonas y rectángulo que las envuelve

Cuando se obtienen todas las regiones que cumplen las condición de tamaño se procede a discriminar las que son de color negro, ya que los objetos colocados sobre la mesa se ven de color blanco y se calcula el punto de interacción de las regiones blancas. El punto de interacción es la zona de la mano con la que se intenta interaccionar con los elementos, ya que no se usa toda la mano, sino que tiende más hacia los dedos que hacia la palma. Este punto siempre se encontrará más alejado con respecto a los laterales de la mesa ya que es la zona en la que se sitúa el usuario intentando ir siempre haica el centro. El punto de interacción depende de determinados factores: el cuadrante de la superficie de *tabletop* dónde se encuentre la mano, el centro del *blob* y las esquinas *blob*, ya que su área viene dada por un rectángulo envolvente.

El cálculo se realiza tomando el centro del *blob* y una esquina *blob* que depende del cuadrante del *tabletop* en el que se encuentre. Una vez elegida la esquina que corresponde se calcula el punto medio desde el centro del blob hacia esa esquina. En la Figura 4.3 se observa la esquina que se debe escoger según el cuadrante dónde se encuentre el *blob*.

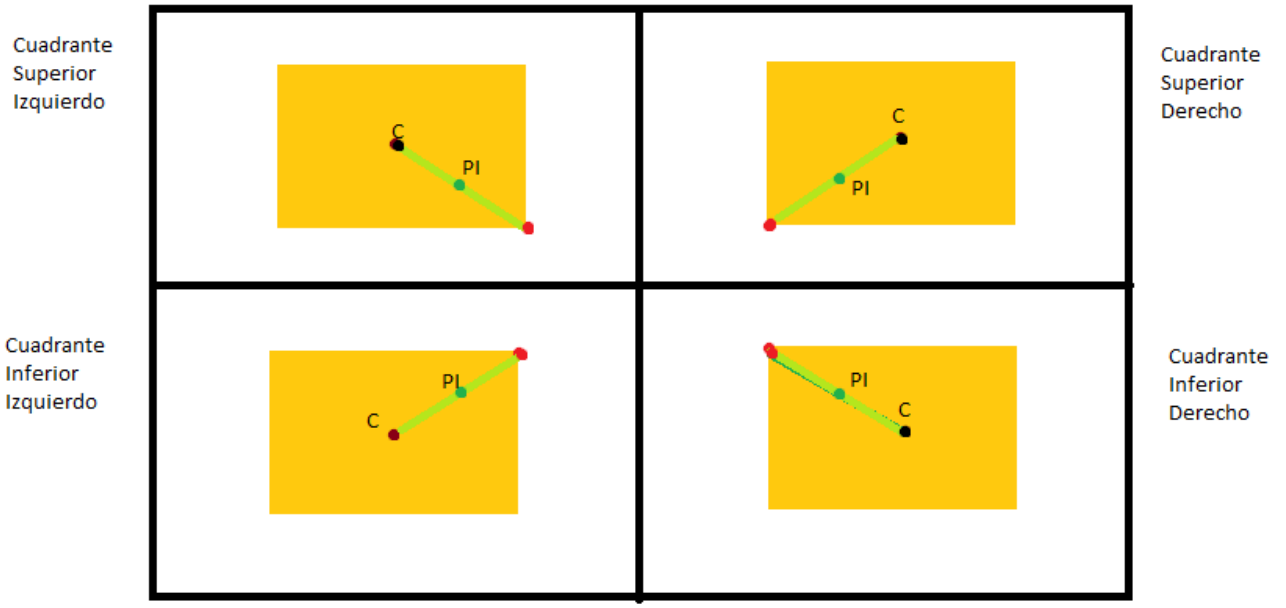

Figura 4.3 Elección de esquina según el cuadrante

Una vez obtenido este punto se almacenará como la posición del *blob* mano y se pasará a analizar otros *blobs*. En la Figura 4.4 se observa la detección de los *blobs* mano y sus puntos de interacción, encontrándose éste en la esquina inferior derecha debido a que la mano estaba en la región superior izquierda.

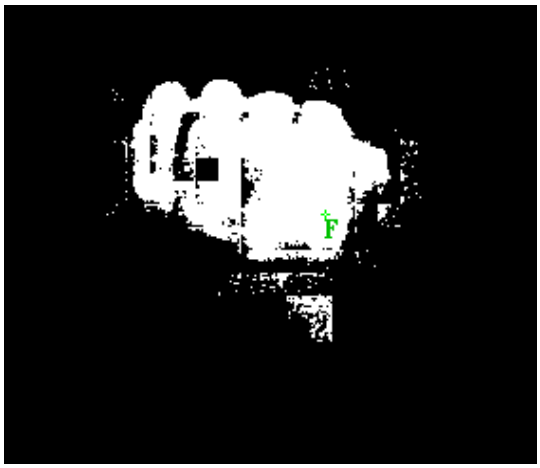

Figura 4.4 Detección de los *blobs* y sus puntos de interacción

El mismo algoritmo se usa para detectar juguetes planos, ya que el *framework* detecta las regiones blancas que se encuentren dentro del rango, pero no sabe si es una mano o un objeto. Esto se aprovecha para poder detectar juguetes con un tamaño específico, siempre y cuando no se necesite identificar específicamente que tipo de juguete es. En la Figura 4.5 se puede ver como 2 objetos cuya parte inferior es blanca son

detectados por ReacTIVision, aunque no se posee ninguna información sobre que tipo de elemento es, sólo su posición.

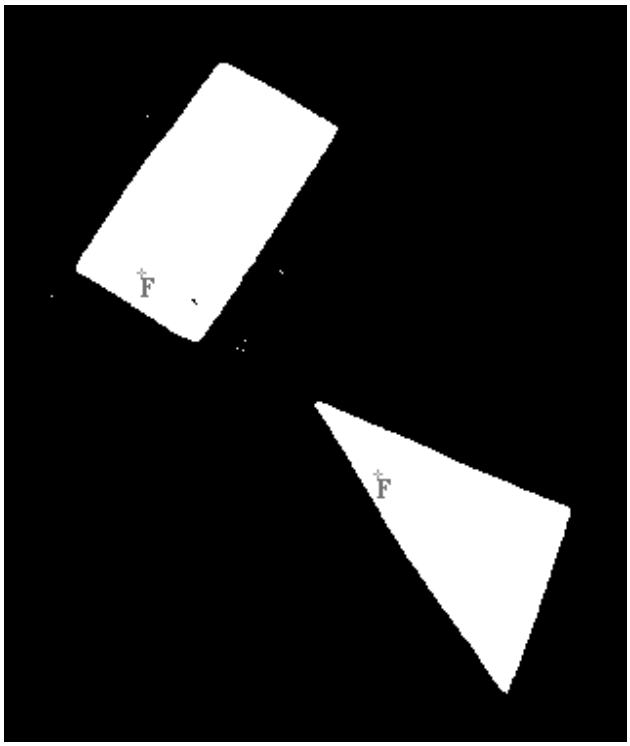

Figura 4.5 Detección de los un objeto plano

4.3 Cálculo de la orientación de blobs

Una vez detectados *blobs* de manos y objetos planos se puede obtener la orientación de los mismos para aportar esa información y utilizarla en juegos. El cálculo de la orientación se basa en las propiedades de los descriptores de forma de una imagen y en las fórmulas para calcularlos [DIMG]. Para obtener la orientación se recorren, uno a uno, todos sus píxeles, tomando como punto de referencia el centro del *blob*. A cada píxel se le asigna un valor (0 para los negros y 1 para los blancos) y se considera la imagen como un sistema de coordenadas, usando el centro del *blob* como centro del sistema. Para calcular la orientación es necesario obtener los momentos centrales de la imagen que son *momentos* independientes de la posición y se basan en el centro y la forma del *blob*. Para calcular los momentos centrales se aplica la siguiente fórmula:

$$\mu_{p,q} = \Sigma(i - i_c)^p(j - j_c)^q$$

Para la fórmula anterior:

- i, j: coordenada del píxel en la pantalla.
- i_c, j_c: coordenada del centro del *blob.*
- p, q: tipos de momento calculado.

Para la obtención de la orientación se han de calcular los momentos de orden 2 de la imagen: μ_{11}, μ_{20}, μ_{02}, quedando de la siguiente forma:

$$\mu_{11} = \Sigma(i - i_c)(j - j_c)$$
$$\mu_{20} = \Sigma(i - i_c)^2$$
$$\mu_{02} = \Sigma(j - j_c)^2$$

El cálculo bien se realiza contabilizando únicamente los píxeles de color blanco, ya que los negros no forman parte del objeto o mano que se quiere detectar. El algoritmo consiste en ir verificando el color de cada píxel del *blob* y, si es blanco, se realizan las operaciones anteriores. Una vez se han obtenido los resultados de las fórmulas anteriores se procede a calcular la orientación a partir de la siguiente fórmula:

$$\Phi = 0.5 \tan^{-1}(2\mu_{11}/\mu_{20} - \mu_{02})$$

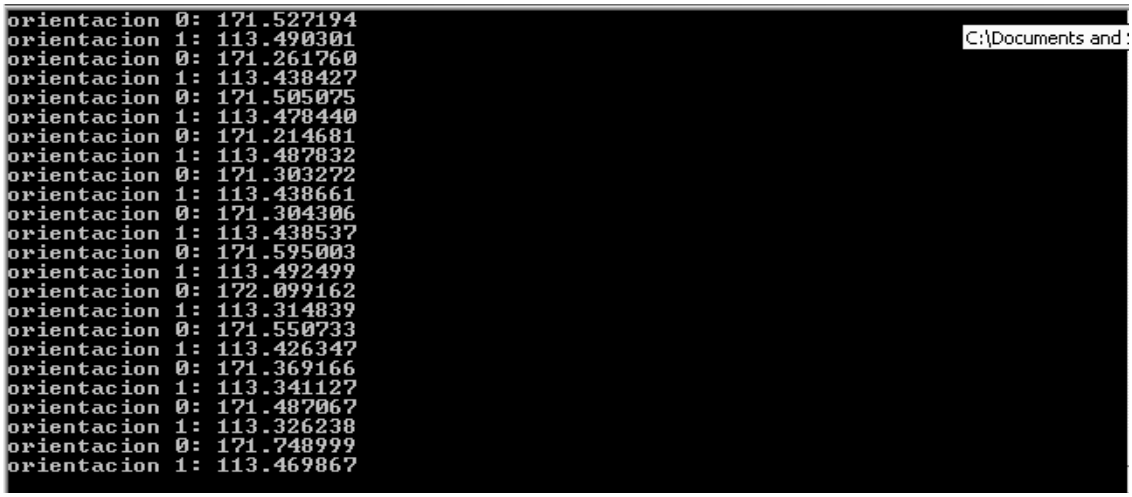

Figura 4.6: Cálculo de la orientación de *blobs.* En cada *frame* que se analiza obtenemos la orientación de los 2 *blobs* de la Figura 4.5

En la Figura 4.6 se observan las orientaciones de los 2 *blobs* de la Figura 4.5. Sin mover los objetos, cada una de las orientaciones se mantiene constante, aunque con ligeras variaciones en los decimales, debido a que al realizar el cálculo hay algunos píxeles que no se detectan bien en algunos *frames* debido a la umbralización que realiza ReacTIVision.

La orientación obtenida se encuentra acotada entre 0º y 180º, debido a que por el algoritmo de cálculo de orientación usado, considerando los 180º como si fuesen 0º.

Existe una excepción en el cálculo de la orientación para los *blobs* de forma circular. Los círculos no poseen orientación debido a que la operación de resta que se realiza en el denominador da como valor 0, siendo una división por 0 errónea. Esto sucede porque, en un círculo, la suma de los cuadrados de las distancias por filas y por columnas es la misma, ya que tienen el mismo número de píxeles. Como la funcionalidad de detectar *blobs* circulares (*fingers*) ya estaba incluida en la versión inicial de ReacTIVision se ha omitido el cálculo de la orientación para esos casos en concreto, aplicándolo únicamente a las manos y objetos planos. Una vez que se calcula la orientación del *blob*, se procede a comunicar la información para que otras aplicaciones la usen.

4.4 Envío de la información

Una vez que se tienen la posición y la orientación de los *blobs* se procese a enviar la información para ser usada por las aplicaciones. Cómo la estructura de *fingers* y *blobs* es parecida, exceptuando la orientación, se procede a modificar el protocolo de comunicación para poder añadir este parámetro. Se han de modificar tanto el módulo que se encarga de realizar el envío como el que se encarga de recogerlo. En el momento en que se crea el paquete que se va a enviar, se incluye un campo extra que contiene la información de la orientación. Como también hay que poder comunicar la información relativa a los *fingers* se incluye un valor erróneo en ese campo cuando se trate de éstos.

En el módulo que recibe la información también se ha de incluir el tratamiento para el campo de la orientación. En el momento en que se procede a descomponer el paquete para extraer la información se le indica que tiene que leer un campo más, ya que también se envía esa información con los *fingers.*

La separación del método de envío en 2 tipos diferentes, uno para *blobs* y otro para *fingers,* hubiese sido más costoso que la solución obtenida, ya que hubiese hecho falta duplicar todas funciones según cada caso. Modificando simplemente un campo se evita al duplicación de funciones y pese a que se estén enviando valores erróneos, la aplicación cliente no necesita distinguir si la información que le llega es de un *finger* o de un *blob.*

4.5 Configuración de los valores de tamaño

Los *blobs* mano y los objetos planos pueden ser de cualquier tamaño, dependiendo de las manos del niño que lo use, la forma y tamaño del objeto plano, etc. Debido a esto no era factible trabajar con unos tamaños mínimo y máximo fijos, ya que esto obligaría a introducir restricciones en el desarrollo de los juegos y de los juguetes. Por eso, al igual que en el proceso de captura de dibujos, se optó por permitir aumentar o disminuir esos valores para poder adaptarlo a las diferentes necesidades. La implementación se realizó de la misma manera que la configuración del tamaño de recorte de los dibujos. La activación, en este caso, es mediante el pulso de la tecla 'm', porque configura los tamaños mínimo y máximo que puede tener el *blob*.

La configuración de los tamaños posee ciertas restricciones obvias. El valor del tamaño mínimo del *blob* nunca podrá ser mayor que su tamaño máximo, ya que esto llevaría a una incongruencia en la que ningún *blob* sería detectado. De igual manera el tamaño máximo nunca podrá ser menor que el tamaño mínimo del *blob*, ya que llegaría a la misma incongruencia.

Por otra parte el máximo valor que se le puede dar a un *blob* es de 400x400 píxeles, ya que es imposible encontrar un elemento de interacción de mayor tamaño. Los valores por defecto que se han dado son: mínimo 50x50 píxeles y máximo 200x200. Estos valores se dieron al comprobar que eran los límites en los que la cámara podía detectar, tanto un puño de un niño pequeño, como el de un adulto. En la Figura 4.7 se observa el menú de configuración de los parámetros de tamaño para la región de búsqueda.

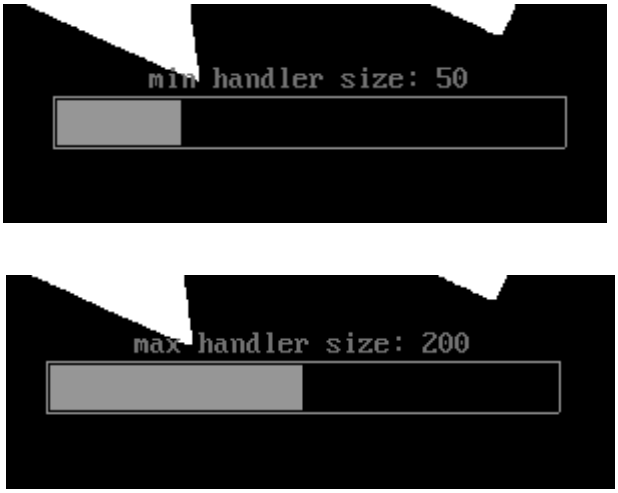

Figura 4.7 Parametrización de los valores de configuración: rango mínimo (pág. anterior) y rango máximo.

Capítulo 5. Resultados

Al finalizar cada una de las implementaciones del proyecto se ha llevado a cabo el desarrollo de un juego para comprobar el correcto funcionamiento de lo desarrollado y ver si cumplía la función de mejora de la interacción para la que se había diseñado.

En este capítulo se presentan los juegos realizados a raíz de las mejoras explicadas en los capítulos anteriores: personalización de *fiduciales,* a través de un juego de asteroides; captura de dibujos, por medio de un juego de pintar dibujos; y detección de *blobs,* mediante un juego de movimiento de imágenes y otro juego de golpear dibujos. Además al final del capítulo también se incluyen unas conclusiones.

5.1 Juego de asteroides (Personalización de fiduciales)

Para realizar este juego se personalizaron 2 *fiduciales* para colocar debajo de 2 juguetes que actúan como naves, colocándose en la base, en su parte delantera. Las naves poseen un pulsador con un *blob finger* en su base, que al ser apretado por el niño se posa sobre la pantalla. Cuando se realiza una pulsación, el juego detecta el *finger* y calcula la distancia al *fiducial* más cercano para saber de cual es el disparo.

En la Figura 5.1 se observa una foto del juguete usado a modo de nave y de su base con el *fiducial*.

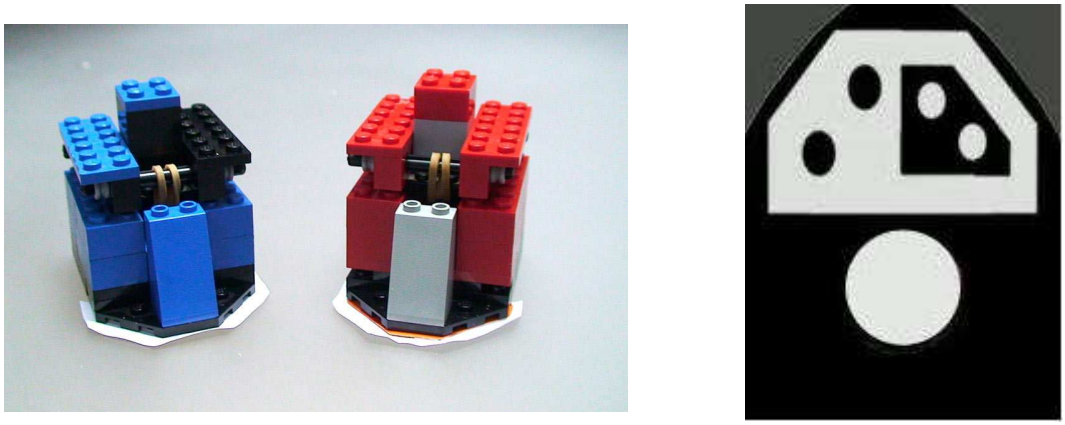

Figura 5.1 Forma del juguete nave (izquierda) y parte de abajo de ésta dónde se observa el *fiducial* añadido.

Debido a la forma de la nave y a la situación del pulsador colocar un *fiducial* con forma totalmente cuadrada hubiese hecho que la nave fuese más grande o que el pulsador se hubiese tenido que colocar en otra parte, haciendo el juguete más grande y su uso más pesado. Para poder incluir el *fiducial* a la base se decidió que no fuese totalmente cuadrado si no que tuviese forma trapezoidal para adaptarse al morro de la nave, de esta forma se aprovecha que el espacio que queda en la nave para colocarlo. Para obtener la información sobre la orientación del disparo, se calcula el vector entre el centro del *fiducial* y el *finger.* El *tabletop* detecta la posición de las naves y, cuando se acciona el pulsador de la nave, se calcula la orientación, permitiendo que el disparo se efectúe en la dirección en la que se encuentra mirando la nave.

En la Figura 5.2 se observa el escenario y modo de funcionamiento del juego. En la superficie del *tabletop* se dibuja un campo de asteroides y se mueven por la pantalla. Cuando un disparo realizado por alguna nave, alcanza uno de los asteroides, éste se subdivide en más pequeños para que tengan que ser destruidos también.

Figura 5.2 Capturas del Juego de asteroides en el que se observa su funcionamiento.

5.2 Juego de pintar dibujos (Tratamiento de dibujos)

Este juego ha sido desarrollado para trabajar en la captura y tratamiento de dibujos. El juego consiste en realizar un dibujo sobre un folio que tiene impreso un *fiducial*, concretamente uno personalizado con forma de lápiz. Que el *fiducial* tenga esta forma indica al niño que debe de realizar un dibujo o algo gráfico en la zona del papel. En la Figura 5.3 se puede observar el folio con un *fiducial* y un dibujo realizado.

Figura 5.3 Folio con *fiducial* y dibujo.

Una vez que el *framework* ha capturado el dibujo, el juego se encarga de leerlo y proyectarlo sobre la superficie del *tabletop*. El juego permite cargar hasta 10 dibujos, iguales o diferentes. Cuando se realiza la carga de la imagen en el juego, se le realizan modificaciones: aplicación de transparencias a todas las zonas en blanco para poder visualizar los dibujos o el fondo que haya detrás, y la inversión del color del contorno para volverlo blanco para que haya contraste con el fondo negro del juego. Una vez que el dibujo es proyectado en la superficie va desplazándose aleatoriamente por ésta. En la Figura 5.4 se observa un dibujo capturado por la mesa y cargado en la aplicación del juego.

Figura 5.4 Imagen de la aplicación. El mismo dibujo ha sido cargado 2 veces en el juego y se han aplicado las transformaciones de color y transparencia. También se ve que se han distribuido de forma aleatoria por la pantalla.

Una vez se tengan cargadas todas las imágenes que se deseen se procede a interactuar con ella mediante el uso de juguetes de colores con *fiduciales*. Cada vez que un *fiducial* se coloque encima de una imagen en el *tabletop,* ésta se pintará de un color u otro, dependiendo del color de juguete al que esté asociado. En la Figura 5.5 se observa la misma imagen coloreada de distintas formas y los *fiduciales* que activan los colores.

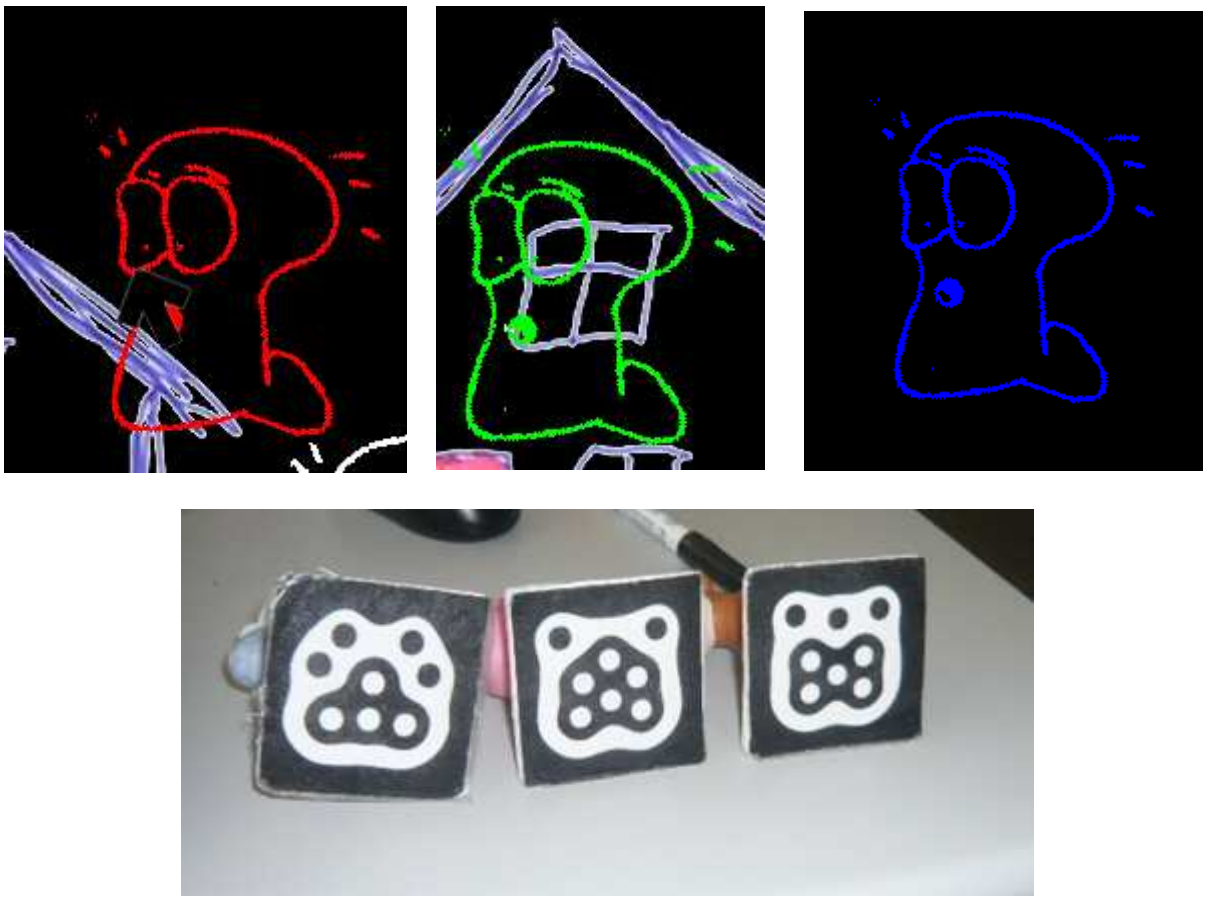

Figura 5.5 Imagen coloreada y *fiduciales* que activan los colores. Rojo (*fiducial* de la izquierda), verde (*fiducial* del centro) y azul (*fiducial* de la derecha).

5.3 Juegos de desplazamiento y golpeo con las manos (detección de blobs)

Para demostrar el funcionamiento de la detección de los *blobs* se han creado 2 juegos. El primer juego consiste en una ampliación del juego de captura y tratamiento de dibujos. Además de lo que realizaba anteriormente se le ha incluido la posibilidad de desplazar por la pantalla los dibujos capturados o los que vienen incluidos (árbol y niña). Gráficamente el juego posee un cursor que se posiciona en la zona dónde se coloca la mano. Una vez que se toca un dibujo o cualquiera de los elementos movibles la imagen se desplazará hacia dónde vaya el *blob.* Si mientras se está desplazando un dibujo se pasa cerca de otro, este también comenzará a seguir al *blob*. En el momento en que se retira la mano de la superficie las imágenes que se encontraban siendo movidas se quedarán quietas en el último sitio y el cursor se desplazará al lugar dónde de detecte otro *blob.*

En la Figura 5.6 se observan 2 imágenes: en la primera se selecciona el dibujo mediante el cursor y en la segunda se observa el desplazamiento de los objetos.

Figura 5.6 Captura del juego de desplazar imágenes. En la imagen de arriba se observa como el cursor se coloca sobre uno de los dibujos. En la imagen de abajo se observa como se ha movido el dibujo, además del árbol.

El segundo juego es una aplicación que muestra de manera aleatoria, tanto en posición como en el tipo, las imágenes capturadas para que el niño las golpee y se eliminen, teniendo que hacerlo en un tiempo aleatorio antes de que desaparezca. Primero se debe introducir un dibujo para que la aplicación empiece a mostrar imágenes. Cuando se golpea una imagen ésta se marca como tocada, dibujando un efecto de golpe y desaparece, teniendo que esperar a que vuelva a surgir en otra posición de la pantalla. Durante el desarrollo del juego se pueden introducir más imágenes provocando que, cuando se detecte el folio, el juego se pare hasta que se capture el nuevo, pidiendo al usuario después que quite el papel. Una vez completado la captura del dibujo se vuelven a mostrar otra vez los dibujos, incluyendo los nuevos que se hayan introducido. En la Figura 5.7 se muestran imágenes de el proceso de captura de imágenes durante el desarrollo del juego y en la Figura 5.8 se muestra la interacción de los usuarios con la mesa para jugar.

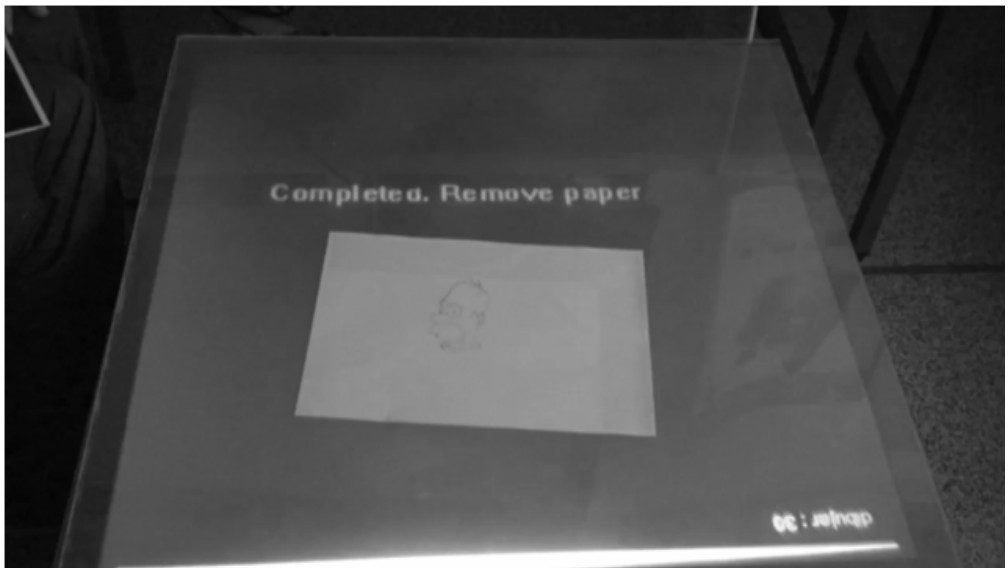

Figura 5.7 Proceso de escaneo de una nueva imagen. Cuando el dibujo se ha terminado de capturar muestra por pantalla un mensaje indicando que ya se puede quitar el folio del *tabletop*.

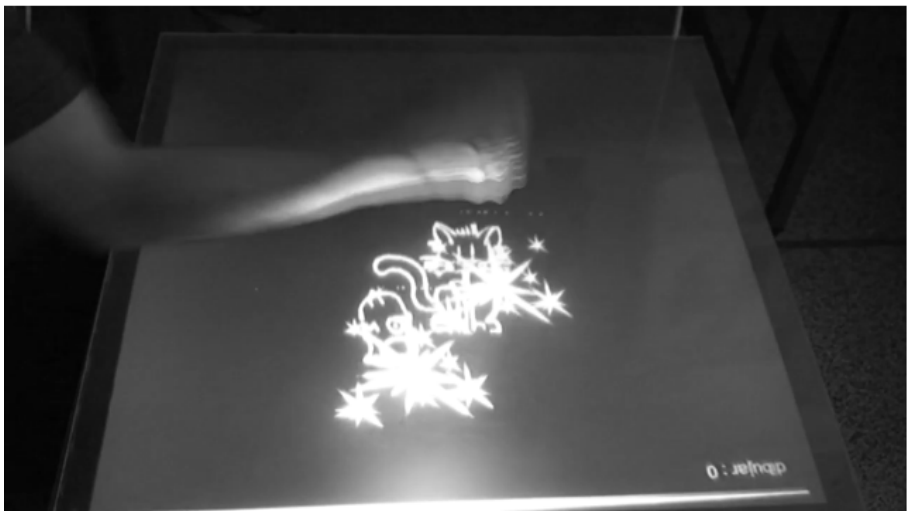

Figura 5.8 Captura del juego. En esta imagen se muestra el proceso de juego, observándose que cuando se golpea una imagen dentro del tiempo límite se dibujan efectos de golpe.

5.4 Conclusiones

Con la creación de estos juegos se ha demostrado que todos los objetivos que se perseguían en el proyecto se han cumplido. Además del objetivo que se quería demostrar con cada juego también se incluyen los anteriores: en cada juego se incluye la personalización de los dibujos y en los juegos de interacción mediante las manos se incluyen también la captura de dibujos. Sin embargo no es necesario para otros juegos que se puedan realizar que se utilicen todas las opciones.

Capítulo 6. Conclusiones

Una vez finalizado el proyecto y obtenidos los resultados se procede a realizar un breve análisis del trabajo realizado en el que se incluye: el cumplimiento de los objetivos principales, valoración, opinión y experiencias conseguidas y posibles mejoras o partes que incrementar del proyecto.

6.1 Cumplimiento de los objetivos principales

En los requisitos del proyecto se especificaban los objetivos principales que se debían cumplir para considerarlo acabado. Como se ve en el desarrollo y en los resultados los objetivos se han cumplido :

- Se han creado *fiduciales* personalizados a los que es posible dar forma, con limitaciones, para que se adapten a los objetos o que posean un significado para el usuario.

- Se ha logrado capturar un dibujo en papel colocado sobre la pantalla pudiéndose además configurar el tamaño de la zona a capturar, rotar la imagen para que sea lo más horizontal posible y limpiar el dibujo para eliminar zonas muy grandes de color negro.

- Se ha conseguido reconocer y detectar manos y objetos planos para interactuar con la mesa, obteniendo su orientación y pudiendo configurar los tamaños que debe detectar.

Además de cumplir los objetivos, todo lo que se ha desarrollado se ha incorporado en diferentes juegos para demostrar su correcto funcionamiento.

6.2 Valoración, opinión y experiencia conseguidas

Desde mi punto de vista personal, este proyecto me ha dado la posibilidad de estudiar y trabajar con la visión por computador y la posibilidad de trabajar con elementos de interacción diferentes a los que existen habitualmente en las aplicaciones. El trabajar con un *framework* que ya tenía numerosas opciones implementadas, tener que estudiarlo e implementar opciones que se acoplasen a las que ya habían ha resultado costoso, pero práctico porque he aprendido cosas interesantes relacionadas con el tratamiento de las imágenes y además he visto que mi capacidad de interpretar y entender código ajeno ha mejorado. También el hecho de haberlo realizado en C++ me ha dado más soltura en el dominio de este lenguaje, aprendiendo cosas que antes no sabía, como la existencia de elementos lista, así como librerías existentes para el tratamiento del vídeo.

Poder probar las mejoras manualmente, interaccionando, como lo harían los propios niños, considero que aporta conocimiento más amplio de lo que se está ejecutando y permite descubrir con más facilidad los errores que puedan surgir y tener idea sobre lo que se puede mejorar con el uso. El hecho de no tener que realizar simplemente un algoritmo que resuelva un problema, si no tener en cuenta que es lo que el usuario va a

realizar, supone también un reto. El niño quiere interactuar y jugar de la forma en la que lo hace cotidianamente, por lo que ha habido que adaptarse a su capacidad de manipulación, y no imponerle como debía hacer las cosas, ya que eso hubiese provocado rechazo a usarlo. Algunos de los casos han sido: cálculo del tiempo que había que dejar quieto el folio; ajustarse a lo rápido que mueve las manos y a su respuesta ante los eventos, etc.

El valor que tiene el proyecto es interesante, ya que se han incluido opciones que antes no estaban implementadas en el *framework* para facilitar el acceso a los niños a los sistemas informáticos y para que les resulte atractivo su uso desde pequeños. Estas mejoras pese a que hayan sido desarrolladas para su uso por niños, también pueden ser utilizadas para crear aplicaciones para ser usadas por cualquier tipo de usuario, como por ejemplo interacción con elementos del ordenador a través del *tabletop*.

Por otra parte, hay señalar que algunas de las ideas que se intentaron probar para lograr alguna solución acabaron no siendo posibles. Una fue personalizar *fiduciales* muy alargados ya que por la forma de detección de *fiduciales* en regiones cuadradas provocaba que la proximidad entre ellos no los hiciese reconocibles. También hubo ideas que se tuvieron que enfocar de otra manera como la limpieza de los dibujos: al principio se probó a eliminar los grupos muy pequeños de píxeles que provocan sensación de suciedad en la imagen, pero se vio que algunas partes importantes del dibujo también se eliminaban, ya que la captura que se realiza pierde calidad por la aberración de la lente. Por esto se optó por dejar esas manchas, ya que al cargar la imagen en las aplicaciones no interfiere.

6.3 Trabajo futuro

Como trabajo futuro o mejoras que se podrían incorporar a este proyecto han surgido varias ideas que se comentan a continuación:

- Aumentar las opciones de la estructura de información para poder enviar más información de la que se permite (posición e identificador en la mayoría de los casos). Esta mejora es bastante simple ya que consiste en ampliar la librería de envío de información TUIO para que trate los nuevos datos que se quieran enviar.

- Modificar el algoritmo de cálculo de orientación de *blobs* para que pueda detectar orientaciones de 360º y no sólo de 180º. Esto tendrá sentido para objetos que no posean simetría de eje, ya que, en el caso contrario, no hay forma de distinguir ángulos de orientación de más de 180º.

- Ampliar el sistema de detección de *fiduciales*, para que las regiones de búsqueda no sean cuadradas, sino que se puedan adaptar a formas rectangulares.

- Ampliar el algoritmo de limpieza de suciedad de las imágenes para limpiar de suciedad el dibujo sin afectar al propio dibujo. Sería necesario un algoritmo más complejo para poder eliminar las zonas de suciedad y distinguirlas de las zonas del dibujo, sin eliminar estas últimas.

Anexo 1. Glosario de términos

En este apartado se incluye una lista de conceptos básicos necesarios para la correcta interpretación de algunos de los aspectos del proyecto:

- **Fiducial:** imagen usada para identificar un objeto mediante un sistema de detección visual. Esta imagen puede dar información, sobre el tipo de objeto que es, la orientación y la posición del objeto. La imagen consiste en un conjunto de elementos blancos y negros.

- **Blob**: elemento de una imagen que viene determinado por ser un conjunto de píxeles de un color en una región discreta, rodeada por píxeles de otros colores. En el caso de este proyecto los *blobs* son binarios, o blancos, o negros.

- **Finger:** *blob* con forma circular que representa un dedo apoyado verticalmente sobre la superficie de la mesa.

- **Tabletop:** mesa usada para la interacción persona – ordenador. La mesa consiste en una superficie transparente en la que se depositan los objetos para ser identificados, un proyector para mostrar una imagen con la que interactuar a través de los objetos y una cámara (vídeo, web, …) para detectar los *fiduciales*, *fingers* y *blobs* que se pongan encima de la superficie.
  La imagen capturada por la cámara se procesa mediante un software, *ReacTIVision* en este caso, y la información obtenida se envía, mediante un protocolo de comunicación, TUIO en este caso, a una aplicación que la trate. Esa información se procesa y se envía al proyector para que la plasme sobre la superficie de la mesa.

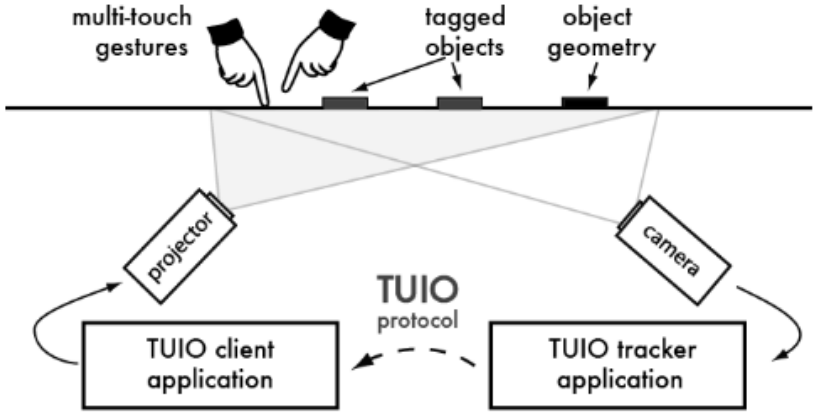

Figura A1.1 Esquema de un Tabletop

- **Framework:** Estructura software de soporte en la que otro proyecto de software puede ser organizado y desarrollado. Esta estructura permite facilitar el desarrollo del software, evitando los detalles de bajo nivel para poder destinar tiempo y esfuerzo a identificar los requerimientos que tendrá lo desarrollado.

- **Multitouch:**Técnica de interacción persona-computador. Esta tecnología consiste en una pantalla táctil que reconoce simultáneamente múltiples puntos de contacto y en un software asociado a ésta que permite interpretar dichas interacciones simultáneas.

- **Frame:** imagen particular que se encuentra dentro de una sucesión de imágenes que componen una animación o video. La sucesión continua de estas imágenes producen sensación de movimiento, provocado por las pequeñas diferencias que hay entre cada una de estas.

- **Umbralización:** acción que consiste en separar, en una imagen en escala de grises, el fondo del objeto, siempre y cuando el fondo y los objetos tengan sus niveles de gris agrupados en 2 niveles dominantes, estableciendo un umbral de separación que puede ser variable.

- **BMP:** Bit Mapped Pictured (mapa de bits). Formato de imágenes que permite guardar imágenes de 24 bits (16,7 millones de colores), 8 bits (256 colores) o menos, sin compresión. Los archivos de mapas de bits se componen de direcciones asociadas a códigos de color, uno para cada cuadro en una matriz de píxeles.

- **Momento:** Parámetros de una imagen que están relacionados con el tamaño, la posición, la orientación y la forma de ésta. Los momentos de denominan de orden p+q, siendo su fórmula:

$$\mathbf{M_{p,q}} = \sum_{1}^{n} \sum_{1}^{m} x^p y^q I(x, y)$$

  Donde *I(x,y)* es el valor de la intensidad del píxel en la posición x,y.

- **XML:** Extensible Markup Language (Lenguaje de marcas extensible). Formato usado para expresar información estructurada a través de etiquetas de la manera más abstracta y reutilizable posible. Una etiqueta consiste en una marca hecha en el documento, que señala una porción de éste como un elemento.

- **Buffer:** Espacio de memoria en el que se almacenan datos para evitar que el programa o recurso que los requiere, ya sea hardware o software, se quede en algún momento sin datos.

- **Display:** Ventana del sistema de detección visual donde se muestra la información de lo que está sucediendo en el *framework* para que el desarrollador o usuario puedan verla.

- **OpenGL**: Siglas de *Open Graphics Library,* una especificación estándar que define una interfaz de programación de aplicaciones que se puede integrar en muchos lenguajes de programación y multiplataforma para crear programas que produzcan gráficos en 2D y 3D. La interfaz consiste en más de 250 funciones diferentes que pueden usarse para dibujar escenas tridimensionales complejas a partir de primitivas geométricas simples, tales como puntos, líneas y triángulos.

Anexo 2. Estructura de ReacTIVision

En este anexo se analiza la estructura de ReacTIVision, explicando en que consiste su funcionamiento y las posibilidades de configuración, el diagrama de las clases que lo componen y un análisis de todas las clases que usa. Durante el análisis todos los elementos modificados o ampliados durante el desarrollo del PFC vienen indicados.

2.1 ¿En qué consiste ReacTIVision?

ReacTIVision es un *framework* de código abierto para el seguimiento y detección de *fiduciales* aplicados en objetos, desarrollado por Martin Kaltenbrunner y Ross Bencina de la universidad “Pompeu Fabra” de Barcelona [RT]. Se diseñó para detectar elementos de *Reactable,* un *tabletop* que funciona como un sintetizador de creación de música, generando sonidos dependiendo de los objetos que se coloquen encima y de su posición *ReacTIVision* funciona de manera independiente a cualquier otro programa, comunicándose con otras aplicaciones a través del puerto UDP mediante mensajes TUIO (ver Anexo 3). En la Figura A2.1 se observa una captura de pantalla de ReacTIVision en el *tabletop* Reactable

Figura A2.1 Captura de pantalla de la ejecución del sistema de reconocimiento de ReacTIVision en el *tabletop* Reactable.

2.2 Funcionamiento de ReacTIVision

ReacTIVision obtiene la información de la cámara y la procesa para realizar la detección de elementos, como *fiduciales* o *fingers,* para enviar esa información a otras aplicaciones.

ReacTIVison trabaja con un motor de tratamiento de superficies SDL (ver Anexo 4) que se encarga de gestionar todo el proceso análisis de la imagen. El *framework* posee varios procesadores de información que tienen la labor de operar con las imágenes procedentes de la cámara. Estos procesadores de información se añaden al motor de tratamiento para que éste gestione cuando se realizarán las acciones de cada uno y en que momento. Cada uno de estos procesadores se encarga de una tarea:

- Ecualizador: Es el procesador que se encarga de tratar la imagen inicial que le envía la cámara.
- Umbralizador: Este procesador se encarga de umbralizar la imagen que proviene de la cámara separando el fondo de los objetos, siendo el fondo negro y los objetos de color blanco.
- Detector de *fiduciales* y *fingers*: Este procesador se encarga de realizar la detección de los *fiduciales* y los *fingers* que se encuentran en las imágenes que se le pasan.
- Calibrador: Este procesador se encarga de corregir la aberración de la cámara que provoca que la geometría de los objetos captados no sea la correcta.
- Servidor de mensajes:Este servidor se encarga de gestionar el envío de la información obtenida después del tratamiento.

La configuración de los procesadores se guarda después de cada uso, de forma que al volver a inicializarlos en usos posteriores mantienen la configuración anterior, siendo el motor el encargado de cargar esa información. El fichero de configuración es un fichero *XML***,** que almacena los valores de las variables que ReacTIVison gestiona.

Cuando todos los procesadores se inicializan y se añaden al motor, el *framework* se pone a trabajar en la detección. En primer lugar se encarga es comprobar que la cámara se encuentra operativa, mostrando error en caso contrario. Después se encarga de mostrar todas las opciones de modificación de los parámetros de los procesadores, además del tipo de cámara usada y de la resolución a la que trabaja.

En la Figura A2.2 se observa la ventana que indica las opciones de configuración de todos los procesadores y de la ventana principal. La configuración de la cámara que se usa es de 640x480 a 30 frames por segundo. Una vez que todo ha sido cargado se procede a ejecutar un bucle que lee la información que llega de la cámara y la trata para mostrar la información en su ventana.

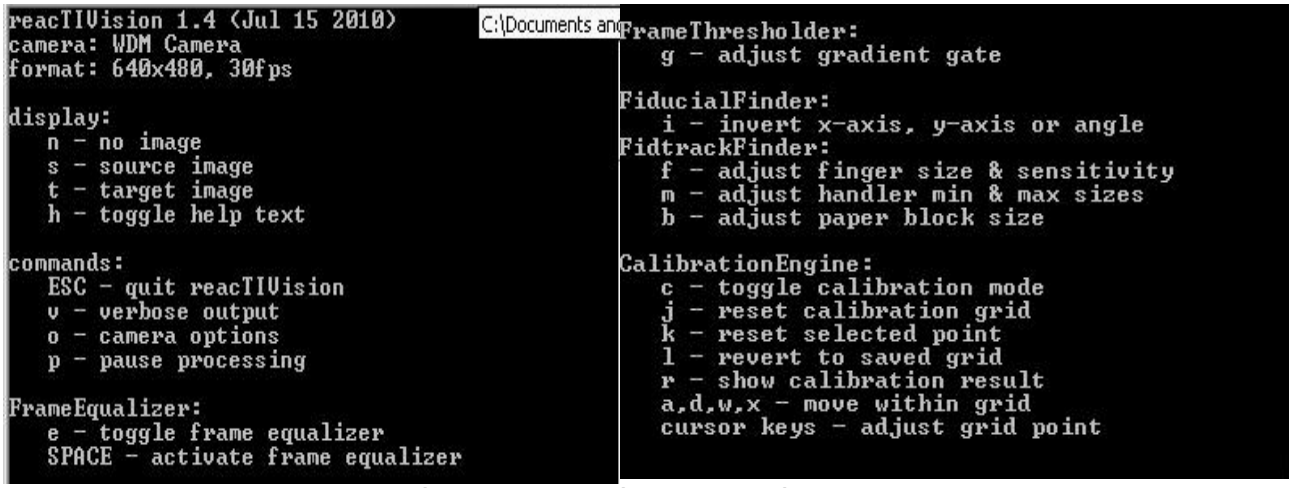

Figura A2.2 Menú de configuración de los parámetros de ReacTIVision

2.2.1 Lectura de la información de la cámara

La gestión de la cámara la realiza un procedimiento que se ejecuta en paralelo al programa principal. Al estar la cámara proporcionando constantemente *frames* al sistema para que los trate no puede dejar de hacerlo en ningún momento, por eso se realiza la ejecución en paralelo. El proceso que se ejecuta en paralelo se encarga en proporcionar continuamente *frames* al *framework* mediante el procedimiento **getFrameFromCamera**. Este procedimiento lee la información del *frame* almacenándolo en un *buffer* de escritura de 3 posiciones, pudiendo almacenar 3 *frames* para que el motor de procesado los vaya tratando. Si en algún momento el *buffer* se encuentra lleno realiza una comprobación para ver si la cámara sigue en funcionamiento, si es así espera 5 segundos antes de intentar enviar más información al motor de procesado para que no se sature. Si, por el contrario, la cámara no responde, el sistema termina el proceso y deja de transmitir información.

El motor de procesado se encarga de leer la información del *buffer* y tratarla mediante los procesadores que tiene asociados para luego mostrarla por pantalla. ReacTIVision puede mostrar las imágenes proporcionadas por la cámara de 3 maneras: la imagen original que recibe de la cámara, la imagen umbralizada mediante el umbralizador o no mostrar ninguna. En la Figura A2.3 se observan las 2 formas diferentes de mostrar la imagen, la original y la umbralizada.

Figura A2.3 Imagen original de la cámara (izquierda) e imagen umbralizada (derecha).

2.2.2 Procesadores de información

Cuando el motor tiene en su *buffer* de lectura un *frame* hace que pase por los diferentes procesadores de información que tiene asociados para tratar la imagen. Estos procesadores se ejecutan siempre siguiendo el orden en que son incluidos en el motor: ecualizador, umbralizador, detector de *fiduciales* y calibrador.

Ecualizador

El ecualizador es el primer procesador en tratar la información de la cámara. Cuando la cámara se encuentra capturando información muchos objetos que se detecten pueden ser del fondo y no ser necesarios, obteniendo falsos datos de objetos que no deberían de estar. Activando el proceso de ecualización se le indica al *framework* que la primera imagen que obtenga será el fondo y que deberá poner toda la imagen como si fuese fondo (de color negro después de la umbralización). Cuando un nuevo objeto se coloque para ser detectado, aparecerá en la pantalla del *framework,* siguiendo el resto de la imagen como si se tratase del fondo. La ecualización se puede activar y desactivar pulsando la tecla 'e' y la imagen que se utilizará como fondo se captura pulsando la barra espaciadora.

Umbralizador

El umbralizador de *frames* se encarga de umbralizar la imagen procedente de la cámara. Utilizando como base la imagen original, la va recorriendo por regiones de 6 x 6 píxeles, obteniendo el máximo y el mínimo valor de cada una de estas regiones. Cuando se tienen todos los valores mínimos y máximos de todas las regiones se procede a umbralizar esas regiones. Usando el dato del valor del gradiente, que se puede aumentar o disminuir mediante la pulsación de la tecla 'g', se comprueba de nuevo cada píxel de la imagen. Si el valor de la diferencia entre el mínimo y el máximo es mayor que el valor del gradiente el píxel se considera fondo y se pondrá de color negro, si por el contrario es menor que gradiente, se comprueba si el valor del píxel es menor que 127, valor medio del total de escala de grises que posee, pasando a blanco si lo cumple o a negro si no.

Detector de *fiduciales* y *fingers*

Una vez la imagen ha sido umbralizada se procede a buscar los *fiduciales* y los *fingers* que se encuentren en ella. Lo primero que realiza es una segmentación de la imagen por zonas de un mismo color. Recorriendo la imagen píxel a píxel se mira el color de cada uno de ellos. Si éste es diferente al anterior, o es el primero, se crea una nueva región del color que sea. Si el siguiente píxel que se analiza es del mismo color que el anterior, se consideran de la misma región, si es de diferente, se crea una nueva. Cuando se han analizado los píxeles de la primera fila de la imagen se procede a analizar los de las siguientes filas. Al igual que en la fila anterior se realizan las mismas comprobaciones añadiendo una al píxel de arriba mirando que si el píxel es igual, se considera que pertenece a la misma región que a la de su superior. Cuando un píxel comparte región con otro anterior, y a su vez con uno superior se procede a fusionar ambas regiones, dando una nueva. En la Figura A2.4 se ve gráficamente como se realiza el proceso de segmentación de la imagen umbralizada.

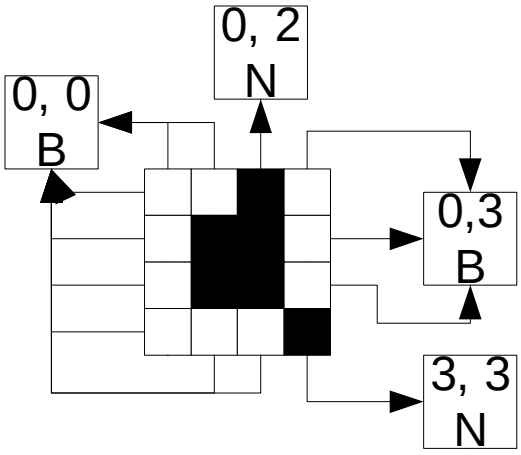

Figura A2.4 Proceso de segmentación

Con la imagen umbralizada y segmentada en regiones se procede a buscar los *fiduciales* que haya en la imagen basándose en el fichero de árboles incluido en la configuración del *framework*. Cuando se tiene la cantidad de *fiduciales* encontrados se procede a ver si son válidos, nuevos o si ya se encontraban en el *frame* anterior. Esta comprobación se hace para crear una nueva información sobre el envío del *fiducial* si no se había detectado antes, o actualizar la información que ya existía, orientación, posición, etc. Como los *fiduciales* pueden haberse desplazado desde el *frame* anterior, por cada uno detectado en el *frame* actual se compara con los anteriores detectados. Al tener los *fiduciales* un identificador comprobaremos sólo aquellos del mismo tipo al que hemos detectado ya que el resto no podrán ser. Cuando todos los *fiduciales* del mismo tipo han sido hallados se calcula la distancia que existe entre el centro del detectado al centro de los otros, si esta distancia es menor que el tamaño medio de un *fiducial,* se considera que el detectado en el *frame* anterior y en el actual son el mismo por lo que se actualizan sus datos, si no, se considera que es un nuevo *fiducial* introducido y se añade como nuevo a la lista de *fiduciales* que han sido detectados.

Una vez se han detectado los *fiduciales* se pasa a la detección de *blobs fingers*. A partir de las regiones segmentadas se procede a buscar las regiones que se encuentren dentro de un tamaño determinado. Este tamaño se puede configurar al pulsar la tecla 'f' del teclado para aumentar o reducir el rango de búsqueda. Una vez obtenidas las regiones se proceden a discriminar las zonas que no posean forma circular, las que no sean de color blanco y las regiones que sean parte de un *fiducial*. Al ser los *blobs fingers* de forma circular y pudiendo estar formados los *fiduciales* también por círculos se ha de evitar que esos círculos sean confundidos con un *blob* circular, por lo que si el *blob* detectado está a menos distancia que la mitad del tamaño aproximado de un *fiducial*, será parte de éste y no deberá contar como un *blob* válido. Una vez que se poseen los *blobs fingers* reales detectados se procede a realizar la misma comprobación que con los *fiduciales* para ver si ese *finger* se encontraba ya detectado con anterioridad o por el contrario es nuevo.

Cuando todo el proceso de detección de elementos se ha llevado a cabo se procede al envío de la información y a la muestra por pantalla de ésta, para los *fiduciales* se mostrará su indicador propio según esté definido el árbol en su fichero; para los *fingers* se indicará mediante el carácter 'F' de color verde el *blob* detectado.

Calibrador

El calibrador es el procesador que se encarga de modificar la imagen recibida para que no existan errores de deformación o para deformarlo según las necesidades del desarrollador. Al activar el calibrador, mediante la tecla 'c', se muestra una rejilla con varios puntos que se pueden mover de posición por la pantalla. El desplazamiento y modificación de los puntos provoca que las zonas de la imagen dada por la cámara que coincidan con esos puntos, se deformarán hasta los nuevos puntos establecidos. La rejilla puede ser puesta otra vez en su forma normal al pulsar la tecla 'j'. En las Figuras A2.5 y A2.6 se pueden observar la rejilla de calibración sin modificar y modificada.

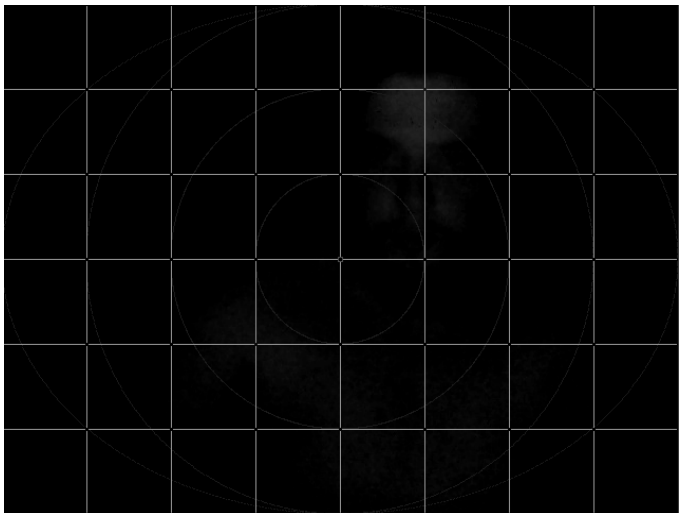

Figura A2.5 Rejilla de calibración. Los puntos de las intersecciones son los que se pueden desplazar para modificar.

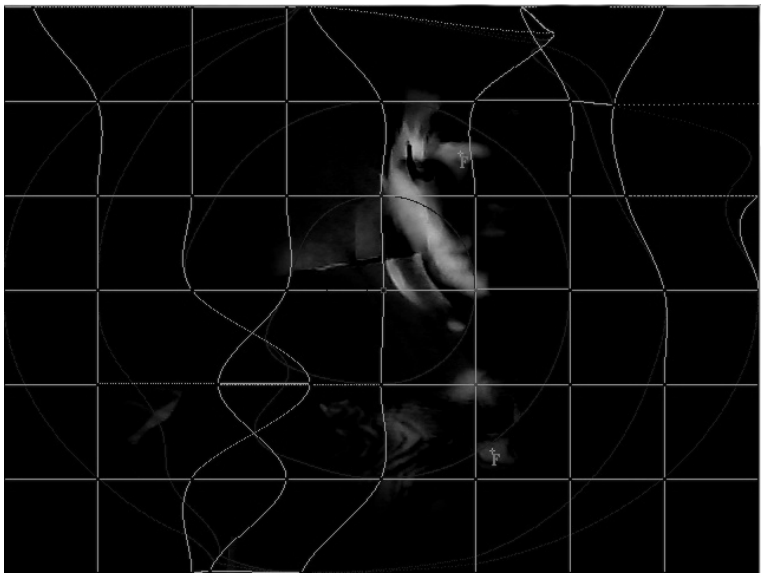

Figura A2.6 Rejilla de calibración con algunos puntos de la imagen deformados

2.2.3 Flujo de datos de ReacTIVision

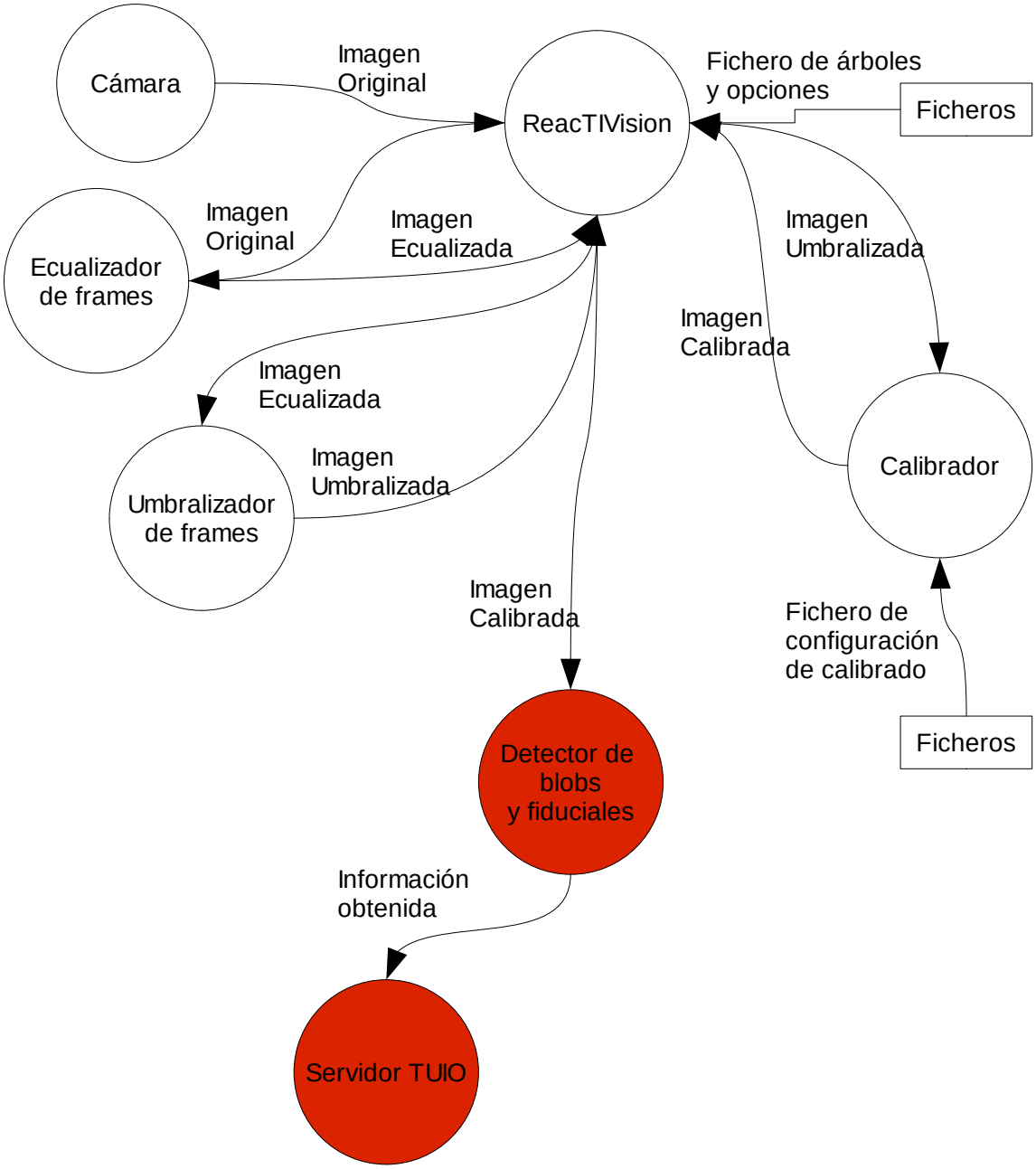

Figura A2.7 Flujo de datos entre los elementos de ReacTIVision.

En la Figura A2.7 se observa el flujo de datos que circula entre los diferentes procesadores y elementos de ReacTIVision. Los elementos que se encuentran coloreados en rojo son los que se han modificado para el desarrollo del proyecto: el detector de *blobs* y *fiduciales* para realizar la captura de dibujos y la detección de *blobs*, y el servidor TUIO para el envío de la nueva información.

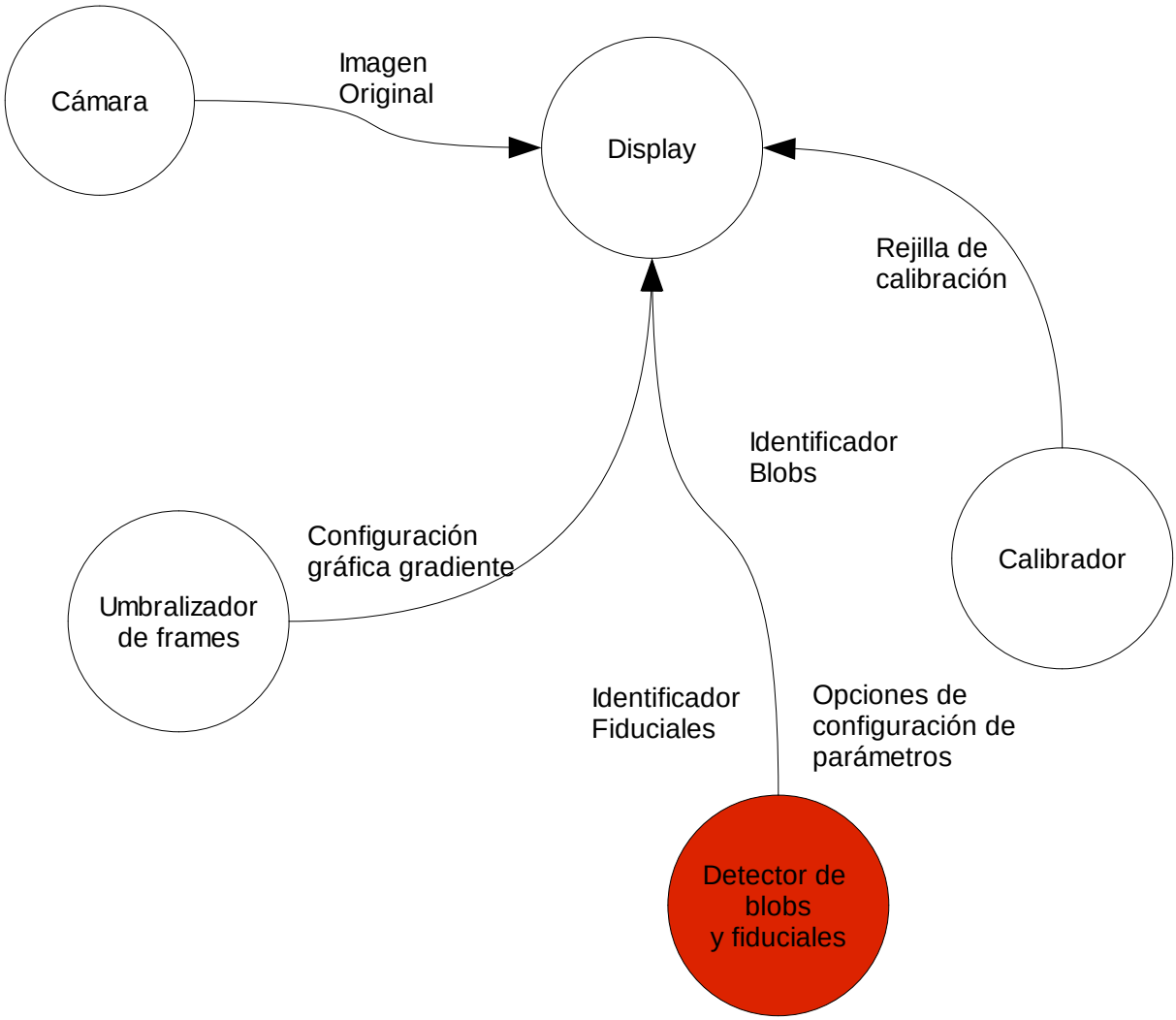

Figura A2.8 Flujo de información entre los elementos de ReacTIVision y el *display* gráfico de información.

En la Figura A2.8 se muestra el flujo de información entre el *framework* y *display.* Los elementos que se encuentran en rojo son los que se han modificado para mostrar nueva información debido a las modificaciones. El detector de *fiduciales* y de *blobs* se ha modificado para que muestre por el *display* los identificadores de los *blobs* detectados, así como las opciones de configuración del tamaño del folio para la captura de dibujos, como las opciones de configuración del rango de detección de *blobs*.

2.3 Diagrama de relación entre clases

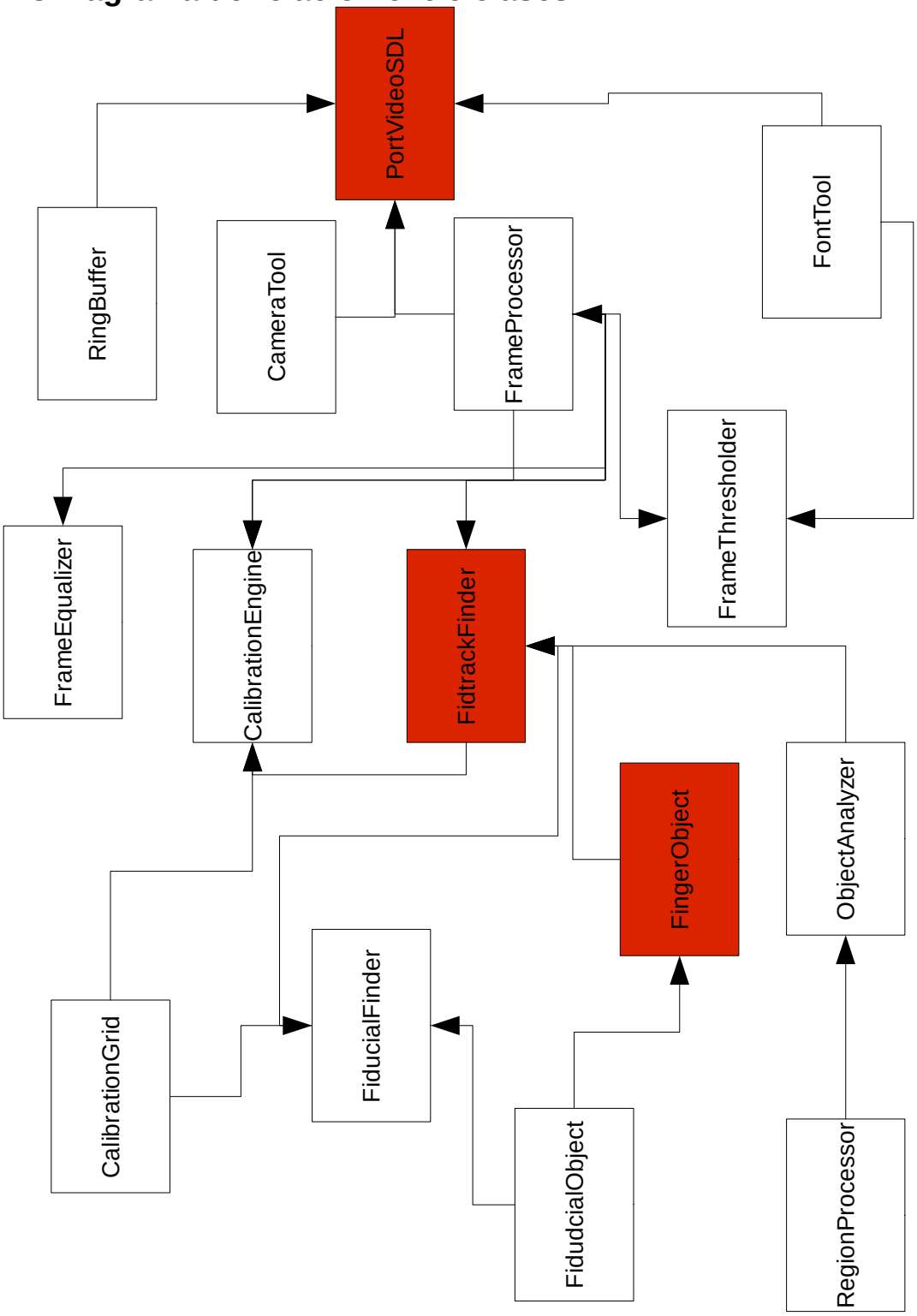

Figura A2.8 Relación entre clases

El la Figura A2.8 se muestra la relación entre las clases de ReacTIVision siendo el origen de la flecha la clase que proporciona información a la clase a la que le llega la punta de la flecha. Las clases que tienen el fondo de color rojo son las que se han modificado en el desarrollo del proyecto. En apartado siguiente se especificarán en cada clase los elementos modificados.

2.4 Diagrama de relación entre clases

En este apartado se incluyen todas las clases que se usan en ReacTIVision, así como sus métodos asociados, sus descriptores y si son clases extendidas de otra. Además se incluyen recuadrados todos los métodos y descriptores modificados o añadidos a cada clase durante el desarrollo del PFC.

FrameProcesor

Clase genérica que contiene los elementos comunes de los procesadores de *frames*.

Descriptores:

Width : Anchura del *frame.*
Height: Altura del *frame.*
SrcBytes: Bytes por píxel en la imagen origen.
DestBytes: Bytes por píxel en la imagen destino.
SrcSize: Tamaño en bytes de la imagen origen.
DestSize: Tamaño en bytes de la imagen destino.
DispSize: Tamaño en bytes del *display*.
Msg_listener: Proceso que gestiona a la clase.

Métodos:

***Init*(int w, int h, int sb, int db)**: inicializa sus descriptores con los valores que se le pasan por paramétros: anchura(w), altura(h), bytes por píxel en la imagen origen(sb) y bytes por píxel en la imagen umbralizada(db).

***Process*(unsigned char *src, unsigned char *dest, SDL_Surface *display)**: realiza un procesado de una imagen dando como resultado una imagen destino e información en pantalla.

AddMessageListener(MessageListener *listener): establece el proceso gestor.

***RemoveMessageListener*(MessageListener *listener)**: eliminar el proceso gestor.

***Finish*():** hace terminar el procesador de frames.

***SetFlag*(int flag, bool value)**: realiza una activación de pulsación de tecla.

***ToggleFlag*(int flag)**: realiza la gestión de teclas.

***GetOptions*()**: devuelve las opciones del procesador.

PortVideoSDL

Clase que actúa como el motor de tratamiento de la información de todo el *framework.*

Descriptores:

Camera_: Dirección de memoria de la cámara.
CameraBuffer_: Buffer donde se almacena lo leido por la cámara.
Camera_config: Nombre del fichero de configuración de la cámara.
Running_, error_, pause_, calibrate_, help_: Booleanos que inidican si el *framework* se encuentra corriendo, si ha habido un error, si está pausado, calibrando o mostrando la ayuda.

Framenumber_: Número de *frame* se que está tratando.
RingBuffer: Buffer que alamacena hasta 3 *frames* indicando cual es el siguiente que debe ser leído.
Current_fps: *Frames* por segundo a los que está trabajando la cámara
Display_lock: Booleano que indica si el acceso al *display* está siendo usado por algún procesador o si se encuentra libre.

Métodos:

***PortVideoSDL(*const char* name, bool background, const char* camera_config):** *C*onstructor de la clase.

***Run*():** Activa el funcionamiento del *framework* ReacTIVision

***Stop*():** Desactiva del funcionamiento del *framework*

***AddFrameProcessor*(FrameProcessor *fp):** Añade un procesador de *frames* al motor.

***MainLoop*():** Ejecuta el bucle principal en el que se realizan todas las acciones de tratamiento de la información de las imágenes. En este método se ha añadido el todo el proceso de tratamiento del dibujo que se va a acaptura: rectificación de la imagen, recorte, limpieza y guardado .

***RemoveFrameProcessor*(FrameProcessor *fp):** Eliminar un procesador de *frames* del motor.

***SetMessage*(std::string message):** Escribe el mensaje pasado para ser mostrado por la consola de comandos.

***DisplayMessage*(const char *message):** Escribe el mensaje pasado en el *display* del *framework.*

***SetDisplayMode*(*Display*Mode mode):** Establece que está mostrando el *display*: la imagen original, la imagen umbralizada o nada.

***GetDisplayMode*():** Devuelve el modo en el que se está mostrando el *display*.

***CurrentTime*():** Devuelve el tiempo que lleva funcionando el *framework.*

***SetupWindow*():** Inicializa la ventana del *display* en la que se mostrará la información de la cámara y de la detección.

***TeardownWindow*():** Elimina la ventana del *display*.

***SetupCamera*():** Inicializa la cámara para que proceda a la captura de imágenes.

***TeardownCamera*():** Deja de usar la cámara.

***InitFrameProcessors*():** Inicializa todos los procesadores de *frames* que tiene añadidos. Si alguno no se puede inicializar, lo elimina.

***AllocateBuffers*():** Establece el tamaño de los *buffers* de origen, destino y *display* y los reserva en memoria.

***FreeBuffers*():** Elimina de memoria el espacio reservado para los *buffers.*

***EndLoop*():** Finaliza la ejecución de los procesadores de información, mostrando por pantalla un mensaje de error si han finalizado por esa razón.

***Process_events*():** Gestiona el manejo de eventos, que en este caso son las acciones que activan la pulsación de las teclas de opciones.

Además de los métodos anteriores se han incluido otras funciones para el tratamiento de la imagen obtenida que permiten obtener el dibujo:

***To_degrees*(float rad):** Transforma a grados los radianes pasados por parámetro.

***Calcular_nueva_pos*(SDL_Rect *r, float rad, int w, int h, int w2, int h2, int af):** Calcula la nueva posición de un punto de la imagen después de realizar un giro a partir de los radianes girados, el tamaño anterior de la imagen (w, h) y el tamaño actual (w2, h2).

***Limpia_bordes*(SDL_Surface *s):** A partir de una imagen dada, s, limpiamos los bordes de la imagen para que no haya zonas en negro muy grandes.

FrameEqualizer

Clase que representa el ecualizador de *frames*. Esta clase es heredera de la clase **FrameProcessor**, por lo que hay procedimientos, funciones y descriptores que también son los mismos para él.

Métodos:

***FrameEqualizer*():** Constructor de la clase que inicializa el procesador.

***Process*(unsigned char *src, unsigned char *dest, SDL_Surface *display):** Procedimiento que realiza la ecualización de la imagen original (src) si se ha pedido que se haga.

***Init*(int w ,int h, int sb, int db):** Inicializa el ecualizador de *frames.*

***ToggleFlag*(int flag):** Realiza la gestión de teclas para la configuración del ecualizador.

***GetState*():** Devuelve si se está ecualizando la imagen original.

FrameThresholder

Clase que representa el umbralizador de *frames.* Esta clase es heredera de la clase **FrameProcessor**, por lo que hay procedimientos, funciones y descriptores que también son los mismos para él.

Métodos:

***FrameThresholder*(short g):** Constructor de la clase que da valor al gradiente con el que se trabajará.

***Process*(unsigned char *src, unsigned char *dest, SDL_Surface *display):** Procedimiento que realiza la umbralización de la imagen original(src) y la almacena ya umbralizada (dest).

***Init*(int w ,int h, int sb, int db):** Inicializa el umbralizador de *frames*.

***DrawGUI*(SDL_Surface *display):** Dibuja en el *display* la información sobre el gradiente actual para que se pueda modificar.

***SetFlag*(int flag, bool value):** Si *flag* es la tecla 'g', se activa o desactiva la modificación del gradiente dependiendo del valor de *value.*

***ToggleFlag*(int flag):** Realiza la gestión de teclas para la configuración del umbralizador.

***GetGradientGate*():** Devuelve el valor del gradiente.

FidtrackFinder

Clase que se encarga de buscar *fiduciales* y *blobs finger* en los *frames.* Esta clase es heredera de la clase **FrameProcessor**, por lo que hay procedimientos, funciones y descriptores que también son los mismos para él.

Métodos:

***FidtrackFinder*(MessageServer *server, const char* tree_cfg, const char* grid_cfg, int finger_size, int finger_sens):** Constructor de la clase que da valor al servidor que envía los mensajes con lo detectado (server), al fichero de configuración de los árboles de los fiduciales (tree_cfg), a la superfice calibrada (grid_cfg), al tamaño de los *fingers* que se van a detectar (finger_size) y a la variación de tamaño que se establece para detectar un *finger* (finger_sens).

***Process*(unsigned char *src, unsigned char *dest, SDL_Surface *display):** Realiza la segmentación y procesado de la imagen umbralizada para detectar *fiduciales* y *fingers* y muestra los datos por el *display*. En este método se han realizado modificaciones para realizar la detección del *fiducial* de captura de dibujo, contabilizar el tiempo que está quiero y enviar la información para que la use la aplicación juego. También se ha realizado la detección de *blobs* realizando la discriminación por rango de tamaño, el cálculo de los momentos para obtener la orientación y la obtención del punto de interacción. Una vez calculado todo se procede al envío de la información en el que se añade el campo orientación para los *blobs*.

***Init*(int w ,int h, int sb, int db):** Inicializa el procesador de detección.

***DrawGUI*(SDL_Surface *display):** Dibuja en el *display* la información sobre el tamaño de los *fingers* que se van a detectar para que se pueda modificar, mostrando el tamaño que deben tener para ser reconocidos. Aquí se han realizado modificaciones para poder mostrar las opciones de configuración del tamaño de la zona de captura (con sus máximos y mínimos) y las opciones del rango máximo y mínimo de la detección de blob. Las opciones son mostradas como una barra horizontal en la que se aumenta o disminuye su valor.

***ToggleFlag*(int flag):** Realiza la gestión de teclas para la configuración del detector. En la gestión se han incluido también las teclas para lanzar la configuración (m y b). Cuando se están modificando los valores de configuración no se puede llamar a ninguna de las otras opciones de modificación de ReacTIVision hasta que no se termine de darles valores.

***GetFingerSize*():** Devuelve el valor del tamaño de los *fingers* que se quieren detectar.

***GetFingerSensitivity*():** Devuelve la variación de tamaño que puede tener un *finger* para que, aunque no sea del tamaño exacto, se pueda detectar.

***GetCalibrationMarkers*():** Devuelve la lista con todos los *fiduciales* detectados en el último *frame* tratado.

***GetCalibrationPoints*():** Devuelve la lista con todos los *fingers* detectados en el último *frame* tratado.

***Reset*():** Borra las listas de *fiduciales* y *fingers* encontrados.

CalibrationEngine

Clase que se encarga de la calibración de la imagen obtenida por la cámara. Esta clase es heredera de la clase **FrameProcessor**, por lo que hay procedimientos, funciones y descriptores que también son los mismos para él.

Métodos:

***CalibrationEngine*(const char* out):** Constructor de la clase dónde se pasa por parámetro el nombre del fichero donde se almacena la información de calibración de ReacTIVision.

***Process*(unsigned char *src, unsigned char *dest, SDL_Surface *display):** Dibuja la rejilla de calibración si se encuentra la opción activada.

***DrawDisplay*(unsigned char* display):** Dibuja la rejilla de calibración.

***Init*(int w ,int h, int sb, int db):** Inicializa el procesador de calibración.

***SetFlag*(int flag, bool value):** Si *flag* es la tecla 'c', se activa o desactiva la modificación de la rejilla dependiendo del valor de *value.*

***ToggleFlag*(int flag):** Realiza la gestión de teclas para la configuración de calibración.

CameraTool

Clase que representa la gestión de búsqueda de la cámara. Sólo contiene un método que representa la búsqueda de todos los dispositivos de cámara que se encuentren disponibles en el sistema para cargar el indicado en el fichero de configuración (config_file). **findCamera(const char* config_file)**

RingBuffer

Clase que representa el *buffer* donde se almacenan los siguientes 3 *frames* que le quedan por trata al *framework*.

Métodos:

***RingBuffer*(int size):** Constructor de la clase que reserva espacio en memoria para los 3 bufferes de lectura a partir del tamaño de un frame (size).

***Size*():** Devuelve el tamaño de un *frame* en bytes: altura * anchura * bytes por pixel.

***GetNextBufferToWrite*():** Devuelve la dirección en memoria del siguiente *buffer* donde se guardará el siguiente *frame* a tratar. Si el *buffer* en donde se ha de escribir está pendiente de leerse, no devuelve ninguna dirección.

***WriteFinished*():** El *frame* ha sido almacenado correctamente para ser leído y avanza al siguiente índice en el que se va a escribir.

***GetNextBufferToRead*():** Devuelve la dirección en memoria del siguiente *buffer* donde está almacenado *frame* para tratarlo en ese momento. Si el *buffer* en donde

se ha de leer está pendiente de escribir, no devuelve nada.

***ReadFinished*():** El *frame* ha sido leído correctamente para ser leído y avanza al siguiente índice en el que está el siguiente *frame* a tratar.

FontTool

Clase que gestiona la escritura de texto por el *display* gráfico de la pantalla.

Métodos:

***Init*():** Inicializa en memoria la fuente que se va a utilizar para escribir por pantalla, ésta viene por defecto.

***Close*():** Libera de memoria la fuente.

***DrawText*(int xpos, int ypos, const char* text, SDL_Surface *display):** Escribe el texto dado por *text* en la posición indicada (xpos, y pos) en el *display* gráfico.

***GetFontHeight*():** Devuelve el tamaño de letra que se está usando para escribir.

***GetTextWidth*(const char *text):** Devuelve la anchura en píxeles del texto (text).

FiducialFinder

Clase que gestiona el buscador de *fiduciales* en un *frame*. Esta clase es heredera de la clase **FrameProcessor**, por lo que hay procedimientos, funciones y descriptores que también son los mismos para él.

Descriptores:

FiducialList: Lista dónde se almacenan los *fiduciales* que se han detectado en el último *frame* tratado.
Session_id: Identificador de sesión que indica en que *frame* ha sido detectado un *fiducial*.
Totalframes: Número total de *frames* analizados.
Grid_config: Nombre del fichero de configuración en donde se almacenan los datos de la rejilla de calibración.
Calibration, show_grid, empty_grid: Booleanos que indican si el *framework* está claibrado, si hay que dibujar la rejilla y si esta está modificada o no.
Tuio_server: Servidor de mensajes TUIO por el cual se enviará la información detectada.
Show_settings: Booleano que indica si se han de mostrar por el *display* las opciones de configuración.

Métodos:

***FiducialFinder*(MessageServer *server, const char* grid_cfg):** Constructor de la clase que recibe por parámetros el servidor de mensajes (server) y el fichero de configuración de la rejilla (grid_cfg).

***DrawObject*(int id, int xpos, int ypos, SDL_Surface *display, int state):** Escribe en el *display* el identificador de un *fiducial* detectado en la posición indicado (xpos, ypos) en el color indicado por el estado (state). Este color puede ser verde si el *fiducial* está bien detectado o rojo si el *framework* no está seguro debido a una mala calidad de la imagen.

***DrawGrid*(unsigned char *src, unsigned char *dest, SDL_Surface *display):** Dibuja en el *display* gráfico la rejilla y distorsiona la imagen original para que los puntos coincidan con la deformación aplicada sobre la rejilla.

***DrawGUI*(SDL_Surface *display):** Dibuja en el *display* la información sobre la inversión en la detección en el eje *x* y en el eje *y* además de en el ángulo.

***ComputeGrid*():** Calcula la matriz de distorsión para aplicar a los puntos de la imagen a partir de los valores de la rejilla modificados.

***SendTuioMessages*():** Se encarga de enviar los mensajes de detección, actualización o borrado de *fiduciales* mediante el protocolo de comunicación TUIO.

FingerObject:

Clase que representa un *blob finger* con sus propiedades. Con la nueva implementación realizad además también representa a los *blobs* mano y objetos planos.

Descriptores:

Alive: Indica si el *blob* finger se encuentra activo en el último *frame* analizado.
Unsent: Indica si el *blob* está siendo detectado, pero todavía no se ha enviado información sobre él.
Session_id: Identificador de sesión del *blob*.
State: Estado del *blob:* añadido, borrado, expirado o vivo.
Smallest_area: mínima área que el *blob* tiene que tener para ser detectado.
Xpos, ypos: posición en pantalla del blob.

Además se ha incluido un campo que contiene la información sobre la orientación.

Métodos:

***FingerObject*(int width, int height):** Constructor de la clase usando como parámetros su altura y anchura (width, height).

***Update*(float xpos, float ypos, float area):** Actualiza la información del *blob* en pantalla: su nueva posición y su nueva área.

***Update*(float xpos, float ypos, float area, float orientacion):** Actualiza la información del *blob*, en la que también se incluye la orientación.

***AddSetMessage*(TuioServer *tserver):** Crea un mensaje para enviarlo mediante el protocolo TUIO a otra aplicación cada vez que el *blob* se actualiza. En el mensaje también se añade la información sobre la orientación del *blob*.

***RedundantSetMessage*(TuioServer *server):** Crea un mensaje para enviarlo mediante protocolo TUIO cuando no se ha modificado ninguno de sus parámetros.

***GetStatistics*():** Devuelve una cadena con toda la información del *blob*.

***CheckStatus*(int s_id):** Devuelve el estado del *blob* indicado (s_id). Este estado puede ser: añadido, borrado, expirado o vivo.

***Distance*(float x, float y):** Calcula la distancia entre un punto de la pantalla (x, y) y el centro del *blob*.

***Reset*():** Inicializa la información del *blob*.

***GetX*():** Devuelve la posición horizontal del centro del *blob*.

***GetY*():** Devuelve la posición vertical del centro del *blob*.

***GetOrientacion*():** Devuelve la posición orientación de *blob*, sólo para *blobs* mano y para objetos planos.

FiducialObject

Clase que representa un *fiducial* y las características de éste.

Descriptores:

Alive: Indica si el *fiducial* se encuentra activo en el último *frame* analizado.
Unsent: Indica si el *fiducial* está siendo detectado, pero todavía no se ha enviado información sobre él.
Session_id: Identificador de sesión del *fiducial*.
Fiducial_id: Identificador del tipo de *fiducial* que es.
State: Estado del *finger:* añadido, borrado, expirado o vivo.
Root_size, leaf_size: Tamaños de los nodos raíz y hojas del árbol del *fiducial.*
Root_colour: Color del nodo raíz del árbol del *fiducial*.
Node_count: Número de nodos del árbol que representa al *fiducial*.
Xpos, ypos: Posición del *fiducial* en pantalla.
Orientation: Orientación del *fiducial*.

Métodos:

***GetAngle*():** Devuelve la orientación del *fiducial*.

***GetX*():** Devuelve la posición horizontal del *fiducial*.

***GetY*():** Devuelve la posición vertical del *fiducial*.

***IsUpdated*():** Indica si el *fiducial* ha sido actualizado durante el anterior *frame.*

***CheckIdConflict*(int s_id, int f_id):** Comprueba si existen conflictos de interpretación entre el *fiducial* que se trata y otro (f_id) en la sesión (s_id).

***FiducialObject*(int s_id, int f_id, int width, int height):** Constructor de la clase a partir de el identificador de sesión(s_id), el tipo de *fiducial* (f_id) que es y su anchura y altura.

***FiducialObject*(int s_id, int f_id, int width, int height, int colour, int node_count):** Constructor de la clase tomando como valores además el color del nodo raíz (colour) y el número de nodos que posee (node_count).

***Update*(float x, float y, float a, float root, float leaf):** Actualización de los valores del *fiducial*.

GetStatistics(): Devuelve una cadena con toda la información del *fiducial*.

***AddSetMessage*(TuioServer *tserver):** Crea un mensaje para enviarlo mediante el protocolo TUIO a otra aplicación cada vez que el *fiducial* se actualiza.

***RedundantSetMessage*(TuioServer *server):** Crea un mensaje para enviarlo

mediante protocolo TUIO cuando no se ha modificado ninguno de sus parámetros.

***CheckRemoved*():** Informa si el *fiducial* ha sido removido de la detección.

***Distance*(float x, float y):** Calcula la distancia entre un punto de la pantalla (x, y) y el centro del *fiducial*.

RegionProcessor

Clase genérica que trata el procesado de regiones.

Descriptores:

Width : Anchura del *frame.*
Height: Altura del *frame.*
SrcBytes: Bytes por píxel en la imagen origen.
DestBytes: Bytes por píxel en la imagen destino.
SrcSize: Tamaño en bytes de la imagen origen.
DestSize: Tamaño en bytes de la imagen destino.
DispSize: Tamaño en bytes del *display*.
Initialized: Indica si el procesador de regiones se ha inicializado.

Métodos:

***RegionProcessor*():** Constructor de la clase que inicializa a 0 todos los descriptores del procesador.

***Init*(int w, int h, int sb, int db):** Establece los valores indicados por los parámetros, poniendo *initialized* a true, y calculando el tamaño de las imágenes origen, destino y del *display*.

***Process*(RegionX *region, unsigned char *src, unsigned char *dest, SDL_Surface *display):** Procesa la imagen origen para dividirla en regiones mediante el tipo de región que se indica. Al ser genérica no realiza nada.

***Finish*():** Finaliza el procesado de regiones.

ObjectAnalyzer

Clase que analiza los objetos por regiones. Esta clase es heredera de la clase **RegionProcessor**, por lo que hay procedimientos, funciones y descriptores que también son los mismos para él.

Métodos:

***Process*(RegionX *region, unsigned char *src, unsigned char *dest, SDL_Surface *display):** Realiza el procesado de regiones indicando en el *display* su contorno mediante un rectángulo blanco.

CalibrationGrid

Clase que gestiona la rejilla de calibración de la imagen origen.

Descriptores:

Width, height: Altura y anchura de la rejilla considerando el número de cuadrados que la componen.

Points_: Vector que almacena los desplazamientos de posición de todos los puntos de calibración de la rejilla.

Métodos:

***CalibrationGrid*(int width, int height):** Constructor de la clase, dando valores del el número de cuadrados que habrá en anchura y altura.

***GetWidth*():** Devuelve el número de cuadrados de anchura de la rejilla.

***GetHeight*():** Devuelve el número de cuadrados de altura de la rejilla.

***Get*(int x, int y):** Devuelve el punto que se encuentra en el cuadrado en la posición x, y.

***Set*(int x, int y, double vx, double vy):** Establece el desplazamiento de la posición vx, vy del punto que se encuentra en el cuadrado x, y.

***IsEmpty*():** Comprueba si los puntos de la rejilla han sido desplazados o no.

***Reset*():** Modifica los puntos de la rejilla para que el desplazamiento que tengan sea 0 y vuelvan a la posición de la rejilla normal.

***Load*(const char* filename):** Carga los datos de la configuración de la rejilla a partir del fichero de configuración (filename).

***Store*(const char* filename):** Guarda los cambios en la configuración de la rejilla en el fichero de configuración indicado (filename).

GetInterpolated(float x, float y): Calcula el punto de interpolación de un punto de la imagen (x, y) al punto del cuadrado de la rejilla donde se encuentra la posición.

Anexo 3. Estructura TUIO

En este anexo se explica en detalle la estructura y el funcionamiento del protocolo de comunicación TUIO. Este protocolo es el que usa ReacTIVision para comunicar la información de lo detectado para que lo usen otras aplicaciones. Se analizará lo siguiente: en que consiste TUIO, su funcionamiento, el formato de los paquetes que envía, la información que se envía y la estructura de los elementos que lo componen. Durante el análisis todos los elementos modificados o ampliados durante el desarrollo del PFC vienen indicados.

3.1 ¿Qué es TUIO?

TUIO es un protocolo de comunicación para superficies *multitouch* tangibles. Este protocolo permite la transmisión de todos los eventos que suceden durante la detección de objetos y la información asociada a lo detectado en la superficie. Los datos de control procedentes de una aplicación de seguimiento son codificados y enviados a otra aplicación cliente que es capaz de decodificar el protocolo para utilizarlos. A pesar de que el protocolo TUIO ha sido principalmente diseñado para realizar la comunicación en superficies interactivas tangibles también se ha usado en aplicaciones de diferentes áreas. El protocolo fue creado por los desarrolladores del *tabletop* Reactable para implementar la comunicación entre la mesa y la aplicación que genera el audio. Su definición se realizó para proporcionar una interfaz de comunicación entre las interfaces tangibles y las capas de aplicación para satisfacer las necesidad de manipulación de objetos sobre la superficie de los *tabletops* [TUIO].

3.2 Funcionamiento del protocolo TUIO

El funcionamiento de la comunicación está basado en el tipo cliente – servidor, en el que el sistema de detección envía mensajes a la clase servidor sobre los elementos detectados y la aplicación actúa leyendo de la clase cliente la información que el servidor le proporciona. Cada vez que se procesa un *frame* y se obtiene la información correspondiente a cada elemento, se procede a comunicar esa información a la aplicación cliente, en caso de que sea necesario.

El protocolo define 2 tipos principales de mensajes: mensajes **SET** y mensajes **ALIVE**. Los mensajes **SET** se utilizan para comunicar la información sobre el estado de los elementos que se encuentran siendo detectados, cada vez que se actualiza alguno de ellos, tales como: posición, orientación, etc. Los mensajes **ALIVE** indican el conjunto de los elementos que se encuentran presentes en la superficie, utilizando para ello un identificador de sesión que es único en cada envío. Para evitar errores procedentes de la pérdida de paquetes en la transmisión se incluyen mensajes implícitos de tipo **ADD** (cuando ese objeto no se encontraba en un envío anterior) y **REMOVE** (se había detectado un **ALIVE** de un objeto que en el envío actual no se ha detectado), permitiendo calcular el tiempo de vida de un objeto a partir de la diferencia entre 2 mensajes **ALIVE** consecutivos. En el caso de que se trabaje con varias aplicaciones a la vez y se desee enviar mensajes específicos independientes, existe un mensaje denominado **SOURCE** que identifica a la aplicación que va dirigido, permitiendo multiplexación ente éstas.

Para enviar toda la información hacia el servidor se utilizan mensajes denominados **FSEQ,** que agrupan toda la información recibida en un paquete al que se le aplica un identificador único correspondiente al *frame* que se ha analizado en ese momento, denominado *fseq*.

Para facilitar que la comunicación sea rápida, ya que el procesado de *frames* está ejecutándose continuamente, el protocolo utiliza el transporte *UDP.* Al utilizar este modo de envío aumenta la posibilidad de que los paquetes enviados se pierdan, por lo que en la implementación del protocolo se hace que se envíe la información sobre todos los elementos detectados, aunque no se hayan modificado, permitiendo no perder información aunque se pierdan paquetes. Aunque la superficie se encuentre en reposo, los mensajes **FSEQ** y **ALIVE** son enviados para que se mantenga la coherencia. Al incluir en cada paquete un campo con la identificación sobre el *frame* del que se trata permite que la aplicación cliente mantenga el orden de la información que va recibiendo. Establecer una conexión TCP aseguraría la llegada de todos los paquetes en orden, pero se produciría un aumento del tiempo de procesado y de envío, siendo que lo que se necesita es rapidez en la comunicación.

3.3 Formato de los paquetes

Para que la información enviada se pueda interpretar con mayor facilidad, los paquetes que se transmiten deben tener un formato específico para que el cliente los pueda tratar y obtener sus datos. En la Figura A3.1 se puede observar el formato de paquete.

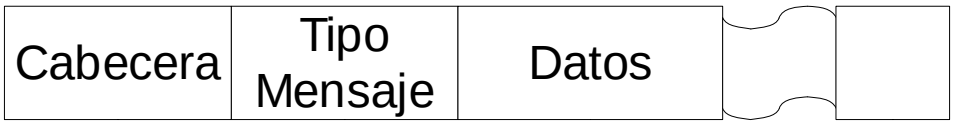

Figura A3.1 Formato del paquete en el protocolo TUIO

Cada paquete comienza con un campo en el que especifica si la información enviada va a tratar sobre **TuioObject** o sobre **TuioCursor.** A continuación se incluye un campo en el que se indica el tipo de mensaje enviado: **SET, ALIVE o FSEQ**. Dependiendo del tipo de mensaje que se haya enviado el siguiente campo de datos contendrá una información u otra:

- Para **SET:** ID de la sesión y la lista de los parámetros para la estructura indicada.
- Para **ALIVE:** Lista de los ID de sesión de todos los objetos que se encuentran activos en el *framework*.
- Para **FSEQ:** Identificador de *frame*.

3.4 Información y atributos de los objetos

En el momento del envío, a cada objeto detectado en el reconocimiento se le asignan unos atributos para incluir en el campo de datos. Los atributos que se incluyen son los siguientes:

- **ID de sesión**: Identificador único para cada objeto que se asigna durante el tiempo que se encuentra siendo detectado.
- **ID de clase:** identificador para indicar que número de *fiducial* se envía.
- **Posición:** Posición en las coordenadas de pantalla del objeto detectado, entre 0 y 1, normalizado para cualquier tamaño de pantalla.
- **Ángulo**: Orientación del objeto. Los objetos *finger* no poseen orientación, entre 0 y 2π.
- **Dimensión:** Altura y anchura del objeto detectado.
- **Vector de velocidad:** Entero que representa la velocidad de desplazamiento en x e y de un objeto basándose en la posición anterior y en la actual.
- **Vector de orientación:** Número en coma flotante que representa la velocidad de rotación de un objeto basándose en la orientación anterior y en la actual.
- **Parámetro libre:** Espacio libre en el paquete para poder enviar información adicional.

3.5 Estructura de los elementos TUIO

En este apartado se analizarán las estructuras de las clase del protocolo TUIO y sus métodos y descriptores. Primero se tratan las clases generadoras de los elementos y después las 2 clases que engloban la comunicación: el servidor y el cliente TUIO.

TUIOTime

Clase que representa el tiempo que ha pasado desde el comienzo de la sesión. El tiempo se representa en segundos y microsegundos.

Métodos:

TuioTime(...): Constructor de la clase. Si no lleva parámetros inicializa el tiempo a 0. Si lleva un parámetro indica los milisegundos con los que se ha de inicializar, dando valor a los segundos y microsegundos. Con 2 parámetros indican los segundos y microsegundos con los que se inicializa.

***Operator*+(long us)**: Suma los microsegundos especificados al tiempo actual.

***Operator*+(TuioTime ttime):** Suma el tiempo de la clase pasada por parámetro al tiempo transcurrido actual.

***Operator*-(long us):** Resta los microsegundos indicados al tiempo transcurrido.

Operator*-(TuioTime ttime)*: Resta el tiempo de la clase pasada por parámetro al tiempo transcurrido actual.

Operator*=(TuioTime ttime)*: Asigna el tiempo de la clase pasada por parámetro a la actual.

Operator*==(TuioTime ttime)*: Compara si el tiempo de la clase pasada por parámetro es igual al actual. Si es cierto devuelve cierto, si no, devuelve falso.

Operator!*=(TuioTime ttime)*: Compara si el tiempo *ttime* es diferente al actual. Si son diferentes devuelve cierto, si no, devuelve falso.

***Reset*():** Restea la cuenta del tiempo a 0.

***GetSeconds*():** Devuelve el número de segundos transcurridos.

***GetMicroseconds*():** Devuelve el número de microsegundos transcurridos.

GetTotalMilliseconds**():** Devuelve el número de milisegundos operando con los segundos y los microsegundos almacenados.

InitSession**():** Inicializa el tiempo de la sesión poniendo a 0 sus valores.

GetSessionTime**():** Devuelve el tiempo que ha pasado desde el inciio de la sesión.

GetStartTime**():** Devuelve el tiempo absoluto que lleva la aplicación corriendo desde anters de inicializar la sesión.

GetSystemTime**():** Devuelve el la hora del sistema en segundos y microsegundos.

TuioPoint

Clase que maneja las posiciones de los objetos, heredando de ella las clases **TuioCursor** y **TuioObject**, que representan los *blobs* y los *fiduciales*.

Descriptores:

> **Xpos, ypos:** Coordenadas X e Y representadas en el rango de 0 a 1, para que sea aplicable a cualquier tamaño de superficie.
> **CurrentTime, startTime:** Descriptores *TuioTime* que representan la última actualización del objeto y el momento en el que se ha inicializado el objeto.

Métodos:

TuioPoint**(float xp, float yp):** Constructor de la clase que inicializa el objeto en la posición indicada y establece el tiempo actual en los descriptores de tiempo (la última actualización y el tiempo de creación son el mismo).

TuioPoint**(TuioPoint *tpoint):** Constructor de la clase que inicializa el objeto a partir de los valores de posición de *tpoint* y establece el tiempo actual en los descriptores de tiempo.

Update**(TuioPoint *tpoint):** Actualiza la posición con los valores de *tpoint.*

Update**(float xp, float yp):** Actualiza la posición con las posiciones pasadas por parámetro.

Update(**TuioTime ttime, float xp, float yp)*****:** Actualiza la clase mediante la nueva posición y el tiempo pasado por parámetro, modificando *currentTime.*

GetX**():** Devuelve la posición en el eje X.

GetY**():** Devuelve la posición en el eje Y.

GetDistance**(float xp, float yp):** Calcula la distancia entre la posición pasada y la del objeto.

GetDistance**(TuioPoint *tpoint):** Calcula la distancia entre el objeto actual y el objeto pasado por parámetro.

GetAngle**(float xp, float yp):** Calcula el ángulo formado entre la posición del objeto y la posición pasada por parámetro. Expresado en radianes.

GetAngle**(TuioPoint *tpoint):** Calcula el ángulo formado entre la posición del objeto actual y la del *tpoint* pasado por parámetro. Expresado en radianes.

GetAngleDegrees**(float xp, float yp):** Calcula el ángulo formado entre la posición del objeto y la posición pasada por parámetro. Expresado en grados.

GetAngleDegrees**(TuioPoint *tpoint):** Calcula el ángulo formado entre la posición del objeto actual y la del tpoint pasado por parámetro. Expresado en grados.

***GetScreenX*(int width):** Devuelve la coordenada X relativa a la anchura de pantalla proporcionada.

***GetScreenY*(int height):** Devuelve la coordenada Y relativa a la anchura de pantalla proporcionada.

***GetTuioTime*():** Devuelve el tiempo de la última actualización del objeto.

***GetStartTime*():** Devuelve el tiempo de la creación del objeto.

TuioContainer

Clase que maneja los atributos de los objetos **TuioCursor** y **TuioObject**. Es heredera de la clase **TuioPoint** por lo que hay funciones y descriptores que son utilizados y definidos.

Descriptores:

Session_id: Identificador de sesión asignado a cada objeto TUIO creado.
X_speed, y_speed: Velocidad en los ejes X e Y.
Motion_speed, motion_accel: Valores del valor de la velocidad y de la acelaración de los objetos. Cálculados a partir de las posiciones anteriores.
Path: Lista de TuioPoints que alamacena todas las posiciones previas del objeto TUIO.
State: Refleja el estado del objeto. Puede ser: `TUIO_ADDED`, `TUIO_ACCELERATING`, `TUIO_DECELERATING`, `TUIO_STOPPED`, `TUIO_REMOVED`.

Métodos:

***TuioContainer*(TuioTime ttime, long si, float xp, float yp):** Constructor de la clase a partir del tiempo actual *ttime*, el ID indicado *si*, y la posición en pantalla. El estado en el que se encuentra es el de `TUIO_ADDED`.

***TuioContainer*(long si, float xp, float yp):** Constructor de la clase a partir del ID indicado si, y la posición en pantalla. El estado en el que se encuentra es el de `TUIO_ADDED`.

***TuioContainer*(TuioContainer *tcon):** Constructor de la clase a partir de otro *TuioContainer*. El estado en el que se encuentra es el de `TUIO_ADDED`.

***Update*(TuioTime ttime, float xp, float yp):** Actualiza la posición del objeto. Mete la nueva posición en la lista y calcula las velocidades y la aceleración. Si *motion_accel* es positiva el estado pasa a ser `TUIO_ACCELERATING`, si es negativa pasa a ser `TUIO_DECELERATING`. Si es igual a 0, `TUIO_STOPPED`.

***Update*(TuioTime ttime, float xp, float xd, float xs, float ys, float ma):** Actualiza los campos del objeto con los parámetros pasados. Además de pasar los datos de anteriores, también se pasan la velocidad en X e Y, además de la aceleración.

***Update*(float xp, float xd, float xs, float ys, float ma):** Ídem anterior, pero sin incluir el campo del tiempo.

***Update*(TuioContainer *tcon):** Actualiza los datos del *TuioContainer* a partir de los datos del parámetro pasado.

***Stop*():** Calcula los valores de velocidad y aceleración sin cambiar las posiciones.

***Remove*(TuioTime ttime):** Modifica el estado a `TUIO_REMOVED` y actualizamos el tiempo.

***GetSessionID*():** Devuelve el ID de sesión del objeto.

***GetXSpeed*():** Devuelve la velocidad en el eje X del objeto.

***GetYSpeed*():** Devuelve la velocidad en el eje Y del objeto.

GetPosition*():* Devuelve la posicion absoluta del objeto en formato *TuioPosition.*

***GetPath*():** Devuelve la lista con las posiciones previas del objeto.

***GetMotionSpeed*():** Devuelve la velocidad que lleva el objeto.

***GetMotionAccel*():** Devuelve la aceleración que lleva el objeto

***GetTuipState*():** Devuelve el estado en el que se encuentra el objeto.,

***IsMoving*():** Comprueba si el objeto se está moviendo. Si su estado es `TUIO_ACCELERATING` o `TUIO_DECELERATING` devuelve cierto, si no, falso.

TuioObject

Clase que representa a los objetos marcadores del sistema de reconocimiento, es decir, los *fiduciales*. Es una clase heredera de *TuioContainer*, por lo que comparte descriptores y métodos de ella. Se añade un estado adicional al *TuioContainer* que simboliza la rotación del objeto, `TUIO_ROTATING.`

Descriptores:

Symbol_id: Número identificador de cada *fiducial* que lo distinge de los demás.
Angle: Ángulo que representa la orientación que posee el objeto.
Rotation_speed, rotation_accel: Valores de la velocidad y la aceleración de la rotación.

Métodos:

***TuioObject*(TuioTime ttime, long si, int sym, float xp, float yp, float a):** Constructor de la clase que le da valor a los campos número de identificador, *sym y* ángulo, *a.* La velocidad y la aceleración de rotación son nulas.

***TuioObject*(long si, int sym, float xp, float yp, float a):** Constructor de la clase que le da valor a los campos número de identificador, *sym y* ángulo, *a.* La velocidad y la aceleración de rotación son nulas.

***TuioObject*(TuioObject *tobj):** Constructor de la clase que copia los valores del objeto pasado por parámetro*.* La velocidad y la aceleración de rotación son nulas.

***Update*(TuioTime ttime, float xp, float yp, float a, float xs, float ys, float rs, float ma, float ra):** Actualiza los campos del objeto con los parámetros pasados. Si el objeto no se encuentra moviendose y existe una aceleración de rotación, el estado cambia a `TUIO_ROTATING.`

***Update*(float xp, float yp, float a, float xs, float ys, float rs, float ma, float ra):** Ídem anterior.

***Update*(TuioTime ttime, float xp, float yp, float a):** Ídem anterior.

***Update*(TuioObject *tobj):** Actualiza los datos del *TuioObject* a partir de los datos del *TuioObject* pasado.

***Stop*(TuioTime ttime):** Calcula los valores de velocidad y rotación sin modificar la posición y la orientación.

***GetSymbolID*():** Devuelve el número identificador del objeto

***GetAngle*():** Devuelve el ángulo de rotación del objeto.

***GetAngleDegrees*():** Devuelve el ángulo de rotación del objeto en radianes.

***GetRotationSpeed*():** Devuelve la velocidad de rotación del objeto.

***GetRotationAccel*():** Devuelve la aceleración de rotación del objeto.

GetPosition*():*** Devuelve la posicion absoluta del objeto en formato *TuioPosition.*

***GetPath*():** Devuelve la lista con las posiciones previas del objeto.

***GetMotionSpeed*():** Devuelve la velocidad que lleva el objeto.

***GetMotionAccel*():** Devuelve la aceleración que lleva el objeto

***GetTuipState*():** Devuelve el estado en el que se encuentra el objeto.,

IsMoving(): Comprueba si el objeto se está moviendo. Si su estado es `TUIO_ACCELERATING, TUIO_DECELERATING` o `TUIO_ROTATING,` devuelve cierto, si no, falso.

TuioCursor

Clase que representa a los detectores de seguimiento de *fingers.* Es una clase heredera de *TuioContainer*, por lo que comparte descriptores y métodos de ella. Esta clase se modificó para almacenar también la orientación de los *blobs*, ya que los *fingers* no tienen orientación.

Descriptores:

Cursor_id: Identificador del cursor individual que se asigna a todos los marcadores.
Orientacion: Orientación del *blob*.

Métodos:

***TuioCursor*(TuioTime ttime, long si, int ci, float xp, float yp, float o):** Constructor de la clase a partir de los parámetros pasados. El ID del cursor viende dado por el parámetro *ci*. También se incluye la orientación dada por *o.*

***TuioCursor*(long si, int ci, float xp, float yp, float o):** Constructor de la clase a partir de los parámetros pasados.

***TuioCursor*(TuioCursor *tcur):** Constructor de la clase a partir de los valores del *TuioCursor* pasado como parámetro.

***GetCursorID*():** Devuelve el identificador del cursor.

***GetOrientacion*():** Devuelve la orientación del cursor.

TuioListener

Clase que proporciona a la clase *TuioClient* una infraestrecutura para tratar los eventos que le llegan sobre lo detectado en el *tabletop*. La clase es genérica y permite definir clases herederas para implementar las acciones que se realizarán cuando les llega un evento.

***AddTuioObject*(TuioObject *tobj):** Método invocado cuando se añade un nuevo *TuioObject* a la sesión.

***UpdateTuioObject*(TuioObject *tobj):** Método invocado cuando se actualiza la información de un **TuioObject** presente en la sesión.

***RemoveTuioObject*(TuioObject *tobj):** Método invocado cuando se elimina un *TuioObject* de la sesión.

***AddTuioCursor*(TuioCursor *tcur):** Método invocado cuando se añade un nuevo *TuioCursor* a la sesión.

***UpdateTuioCursor*(TuioCursor *tcur):** Método invocado cuando se actualiza la información de un **TuioCursor** presente en la sesión.

***RemoveTuioCursor*(TuioCursor *tcur):** Método invocado cuando se elimina un *TuioCursor* de la sesión.

***Refresf*(TuioTime ftime):** Invocado por el *TuioClient* para indicar que ha termiando de recibir un paquete de información.

En las aplicaciones desarrolladas para probar las mejoras implementadas se realizan acciones para comprobar el correcto envío y detección de los elementos. Estas acciones consisten en la ejecución de cógio que se realiza cuando se envía información sobre una detección, actualización o eliminación de un *fiducial* o *blob*.c En el siguiente ejemplo se peude ver que cada vez que se detecta un nuevo *fiducial* la aplicación ejecuta un algoritmo para tratar el objeto según sea su identificador:

```
void TuioListener::addTuioObject(TuioObject *tobj) {
  mesa->tratar_fiducial(tobj->getSymbolID());
}
```

TuioServer

Clase que representa el servidor de mensajes que el *framework* utiliza para enviar la información sobre los *fiduciales* y los *blobs* detectados en el *tabletop*. En esta clase se ha modificado la función de añadir un cursor para poder enviar la información de la orientación de los *blobs*.

Métodos:

***TuioServer*(const char* address, int port):** Crea la conexión *UDP* a partir de la dirección de comunicación y el puerto.

***FreeObjSpace*():** Comprueba si hay sitio en el paquete que se va a enviar para añadir más información sobre *fiduciales*. Si es así devuelve cierto, si no, devuelve falso.

***AddObjSeq*(int fseq):** Crea la cabecera de envío de paquete, incluyendo el identificador de paquete indicado por *fseq*.

***AddObjSet*(int s_id, int f_id, float x, float y, float a, float X, float Y, float A, float m, float r):** Crea un mensaje **SET** a partir de los datos necesarios para enviar un *fiducal:* su identificador de sesión, *s_id*; su identificador de fiducial, *f_id;* la posición absoluta en pantalla (rango 0..1); el ángulo, *a;* la velocidad de desplazamiento en los ejes; la velocidad de rotación, *A*; la aceleración, *m*; y la velocidad de rotación, *r.*

***AddObjalive*(int *id, int size):** Crea un mensaje **ALIVE** a partir de un vector en el que se almacenan los *fiduciales* que se encuentran activos en pantalla y el tamaño del vector.

***SendObjMessages*():** Transmite la información del paquete de objetos, *fiduciales*, a través del *socket* de transmisión, permitiendo el nuevo envío de otro paquete.

***FreeCurSpace*():** Comprueba si hay sitio en el paquete que se va a enviar para añadir más información sobre *blobs*. Si es así devuelve cierto, si no, falso.

***AddCurSeq*(int fseq):** Crea la cabecera de envío de paquete, incluyendo el identificador de paquete indicado por *fseq*.

***AddCurSet*(int s_id, float x, float y, float X, float Y, float m, float o):** Crea un mensaje **SET** a partir de los datos para enviar un *blob*: su identificador de sesión*, s_id;* su posición absoluta en pantalla (rango 0..1); la velocidad de desplazamiento en los ejes y la aceleración, *m*. En esta función se añadió también la información sobre la orientación de los *blobs*, *o*, ya que al estar pensada para tratar *fingers*, que no tienen orientación, esa información no se incluía.

***AddCuralive*(int *id, int size):** Crea un mensaje **ALIVE** a partir de un vector en el que se almacenan los *blobs* que se encuentran activos en pantalla y el tamaño del vector.

***SendCurMessages*():** Transmite la información del paquete de objetos, *blobs*, a través del *socket* de transmisión, permitiendo el nuevo envío de otro paquete.

TuioClient

Clase que representa el cliente de mensajes que se encarga de leer la información enviada por el *framework* para ser usada por la aplicación de la forma necesaria. Cada objeto y cursor que recibe son almacenados en una lista para poder acceder a ellos en otro momento.

Descriptores:

Socket: Variable donde se almacena la dirección de conexión y por donde se intercambia la información.

Métodos:

***TuioClient*(int port=3333):** Intenta realizar una conexión con el puerto especificado. Si no se especifica ninguno, se toma por defecto el puerto 3333. Si no logra realizar la conexión informa por la consola de comandos que no se ha podido.

***Connect*(bool lock = false):** El cliente empieza a escuchar por el puerto UDP para recibir los mensajes. La variable *lock* indica si el cliente escuchará únicamente ese puerto, o creará un hilo de ejecución paralelo por donde escuchará, pudiendo realizar otras tareas.

***Disconnect*():** El cliente deja de escuchar por el puerto UDP para dejar de recibir información.

***IsConnected*():** Comprueba e informa de si existe una conexión activa en el momento o no.

***AddTuioListener*(TuioListener *listener):** Añade un **TuioListener** a la lista de infraestructuras de mensajes que posee. Al añadirla el cliente es capaz de capturar los eventos relacionados con los tipos de datos a recibir.

***RemoveTuioListener*(TuioListener *listener):** Elimina el **TuioListener** indicado por *listener* de la lista dónde los tiene almacenados.

***RemoveAllTuioListeners*():** Eliminar todos los **TuioListener** registrados en la lista dejándola vacía.

***GetTuioObjects*():** Devuelve la lista entera de **TuioObjects** que tiene el cliente almacenados. Antes de acceder bloquea el acceso a ésta para que no se añadan objetos mientras se están leyendo los datos, desbloqueandolo después.

***GetTuioObject*(long s_id):** Devuelve el **TuioObject** que coincide con el ID de sesión indicado, *s_id*. Al acceder bloquea la lista para que no sucedan errores.

***LockObjectList*():** Bloquea la lista para evitar actualizaciones durante el acceso a la lista.

***UnlockObjectList*():** Desbloquea la lista para poder realizar actualizaciones.

***GetTuioCursors*():** Devuelve la lista entera de **TuioCursor** que tiene el cliente almacenados. Antes de acceder bloquea el acceso a ésta para que no se añadan objetos mientras se están leyendo los datos, desbloqueandolo después.

***GetTuioCursor*(long s_id):** Devuelve el **TuioCursor** que coincide con el ID de sesión indicado, *s_id*. Al acceder bloquea la lista para que no sucedan errores.

***LockCursorList*():** Bloquea la lista para evitar actualizaciones durante el acceso a la lista.

***UnlockCursorList*():** Desbloquea la lista para poder realizar actualizaciones.

***ProcessPacket*(const char *data, int size):** Procesa un paquete, obeteniendo los menajes a partir del tamaño de este, *size* y la ristra de datos, data.

***ProcessMessage*():** A partir de los datos de los mensajes que llegan se realiza su procesado. Primero se comprueba que tipos de datos se han enviado para actualizar la lista de objetos o la de cursores. Luego se hace una comprobación sobre el tipo de mensaje enviado: si es de tipo **SET** se comprueba si está en la lista, si no está se crea una nueva clase con la información enviada; si se encuentra en la lista, se actualiza con la nueva información. Si es de tipo **ALIVE** se actualiza la lista de elementos activos con los ID pasados como información. Por último si el mensaje es de tipo **FSEQ** se recorre la lista de objetos existente y se actualizan todos los objetos que se encuentran a partir de la información del estado de éstos. Para la tratar la información de la orientación de los *blobs* no circulares se ha modificado la parte en la que trata los paquetes **SET** para poder incluir la información en su campo. En el caso de que el *blob* sea circular el valor de la orientación será un número que no representa nigún valor (NaN).

Anexo 4. Tratamiento de la librería SDL

En este anexo se analizará la librería gráfica SDL que se utiliza para el tratamiento de la información gráfica. ReacTIVision usa esta librería para realizar el tratamiento de las imágenes que llegan de la cámara web, utilizando sus funciones para recorrer la información de la imagen y poder modificarla.

4.1 ¿Qué es SDL?

SDL son las siglas de *Simple DirectMedia Layer,* una biblioteca multimedia y multiplataforma diseñada para facilitar el acceso de bajo nivel a elementos tales como el audio, teclado, ratón, *joystick;* además de proporcionar soporte al *hardware* 3D mediante *OpenGL* y al *buffer* 2D de vídeo. La biblioteca se encuentra desarrollada en lenguaje C, aunque existen librerías para hacerla funcionar en otros lenguajes, como C++, Ada, etc. Esta biblioteca normalmente es usada para la interfaz gráfica de los reproductores de *MPEG***,** emuladores y videojuegos [VSDL].

4.2 Funcionamiento de SDL

La librería SDL se encarga de proporcionar funciones al programa para realizar las diferentes tareas para la gestión de los elementos multimedia de éste. Las funciones que proporciona se encuentran separadas según la funcionalidad que aporten al sistema: vídeo, tratamiento de eventos, audio, accesos a CD-ROM, tratamiento de hilos paralelos y temporizadores de ejecución. Para acceder a la funciones que se necesiten, es necesario que se inicialicen en algún momento las funcionalidades que se vayan a necesitar, pudiéndose quitar en el momento que se quiera cuando no sea necesario. A continuación se van a comentar las distintas funcionalidades que aporta SDL.

Vídeo

La gestión de vídeo permite la escritura en la dirección de memoria de la pantalla de la imágenes generadas en el proceso. Una de las funcionalidades básicas es el establecimiento de un modo de vídeo para cualquier resolución y profundad de bits, pudiendo realizar un cambio si el *hardware* no soporta el modo empleado. Con el modo de vídeo establecido se permite la escritura en un *buffer* gráfico para poder ser mostrado por pantalla, permitiendo la creación de superficies con atributos como transparencia, colores de los píxeles, etc.

Eventos

El tratamiento de eventos permite la captura de eventos generados por dispositivos de entrada tales como el ratón, teclado, joystick para realizar unas acciones que afecten al programa o para generar salidas solicitadas por el usuario. *SDL* permite que el tratamiento de eventos pueda habilitarse o deshabilitarse durante el funcionamiento de la aplicación, permitiendo que cuando no sea necesario capturar determinados eventos no lo realice y volver a activarlo cuando se pueda. Los eventos que se han capturado, se envían a una cola de eventos interna para mantener el orden en que se capturan.

Audio

La gestión de audio permite la reproducción de audio durante el programa, pudiendo ser el sonido de 8 bits o 16, mono o estéreo y permitiendo la conversión si el formato seleccionado no lo soporta el propio *hardware.* La reproducción del audio se realiza independientemente, en un hilo de ejecución diferente al principal, por lo que no es necesario que sea creado o eliminado por el programador.

CD-ROM

El tratamiento del CD-ROM permite la reproducción de pistas de audio almacenadas en un CD-ROM. Las funciones nos permiten saber el número de dispositivos lectores que hay en el ordenador y realizar las tareas de pausa, parada y activación para cada una de las unidades existentes.

Hilos paralelos

SDL permite la creación de hilos de ejecución paralelos, la posibilidad de implementar semáforos binarios simples para sincronizar los procesos y la creación de barreras de paso. Para la creación de hilos de ejecución paralelos es necesario especificar que función realizará y los datos que necesitará para ejecutarse. A cada hilo de ejecución se le asigna un identificador para tener constancia de su existencia y funcionamiento. A partir de la dirección de ejecución del hilo se pueden eliminar los hilos, obtener sus identificadores, etc. La creación y destrucción de semáforos y barreras se realiza mediante funciones simples.

Temporizadores de ejecución

Estas funcionalidades permiten realizar operaciones temporizadas en la ejecución del programa. Entre las más importantes de sus operaciones se encuentran las opciones de obtener el número de milisegundos transcurridos desde la inicialización de la librería *SDL,* la posibilidad parar la ejecución durante un tiempo especificado y la posibilidad de lanzar la ejecución de un hilo cuando hayan pasado un tiempo especificado.

4.3 Estructuras y funciones de la librería *SDL*

En este apartado se analizan las estructuras y las funciones que proporciona la librería SDL, comentando únicamente los elementos que se han usado durante el desarrollo del proyecto. Los elementos que se han usado en el proyecto son: las funcionalidades de vídeo, la gestión de eventos y las operaciones generales.

Operaciones Generales

***SDL_Init*(Uint32 flags)**: inicializa la librería SDL y los subsistemas indicados por parámetros. Estos subsistemas pueden ser: `SDL_INIT_TIMER` (temporizador), `SDL_INIT_AUDIO` (audio), `SDL_INIT_VIDEO` (vídeo), `SDL_INIT_CDROM` (CD-ROM), `SDL_INIT_JOYSTICK` (joystick), `SDL_INIT_EVERYTHING` (inicialización de todo lo anterior), `SDL_INIT_NOPARACHUTE` (ejecuta el sistema sin que reaccione a elementos fatales) y `SDL_INIT_EVENTTHREAD` (inicializa el gestor de eventos en un hilo a parte).

***SDL_InitSubSystem*(Uint32 flags):** inicializa los subsistemas anteriores después de haber inicializado la librería. Se usa para cargar funciones en algún momento determinado.

***SDL_QuitSubSystem*(Uint32 flags):** elimina los subsistemas que han sido cargados cuando no sean necesarios.

***SDL_Quit*():** elimina todos los subsistemas cargados, descarga la librería de la memoria y libera los recursos usados.

***SDL_WasInit*():** proporciona la información sobre que subsistemas se encuentran inicializados.

***SDL_GetError*():** devuelve una cadena con el último error interno de SDL detectado.

***SDL_SetError*(const char *fmt, ...):** establece una cadena como un error interno de SDL.

***SDL_Error*(SDL_errorcode code):** establece un error según un código interno: `SDL_ENOMEM` (desbordamiento de memoria), `SDL_EFREAD` (error de lectura en memoria), `SDL_EFWRITE` (error de escritura en memoria), `SDL_EFSEEK` (error de búsqueda en memoria).

***SDL_ClearError*():** elimina toda la información sobre el último error detectado. Útil si el error ha sido capturado por un gestor de eventos.

Vídeo

En el tratamiento del vídeo se usan superficies denominadas SDL_Surfaces que son las que almacenan la información del contenido gráfico de las imágenes. Una de las superficies tendrá que ser en la que se pinte toda la información que corresponde a la pantalla.

Estructuras:

SDL_Surface: estructura que representa una zona gráfica en memoria que se puede ser modificada.

> **flags:** opciones aplicables a la superficie.
> ***format:** formato de los píxeles de la estructura.
> **w, h:** altura y anchura de la superficie.
> **pitch:** tamaño de la superficie en bytes.
> ***pixels:** dirección donde se encuentran los píxeles que forman la superficie.
> **clip_rect:** área rectángular asociada a la superficie.

La dirección de los píxeles recorre la imágen por filas dándo la información sobre el valor de cada uno de los píxeles. Cada píxel tiene un valor que va asociado a su vez a una paleta de colores, pudiendo ser el mismo valor un color diferente dependiendo de la paleta que se use y los colores que se quieran usar.

SDL_Rect: área rectangular de una superficie que contiene la posición respecto a la *SDL_Surface* asociada y su tamaño dentro de ella.

> **x, y**: posición de la esquina superior del rectángulo en la superficie.
> **w, h:** anchura y altura de la superficie.

SDL_Color: estructura que representa un color.

r, g, b: variables de 8 bits que representan la cantidad de rojo, verde y azul que tiene el color específicado.

SDL_Palette: estructura que representa una paleta de colores de formato de 8bits.

ncolors: número de colores que forman la paleta.
***colors:** dirección donde se encuentan los colores que forman la paleta de colores.

SDL_PixelFormat: formato que poseen los píxeles que componen una superficie.

***palette:** paleta de colores asociada a la superficie.
BitsPerPixel: número de bits usados para representar un píxel en una superficie.
BytesPerPixel: número de bytes usados para representar un píxel en una superficie. No tiene que ser equivalente a *BitsPerPixel*.
Alpha: valor de transparencia para la superficie.
Rloss, Gloss, Bloss, Aloss: pérdida de precisión de cada coomponente de color.
Rshift, Gshift, Bshift, Ashift: desplazamiento del valor de cada color que compone el píxel.
Rmask, Gmask, Bmask, Amask: máscara binaria usada para recuperar los valores individuales de color.

Funciones:

***SDL_SetVideoMode*(int width, int height, int bpp, Uint32 falgs):** crea una superficie de vídeo con el tamaño indicado en altura y anchura y en bits por píxel. La función devuelve la SDL_Surface de la zona de memoría de vídeo. Las opciones más comunes usadas son: `SDL_SWSURFACE` (crea la superficie en la memoria del sistema), `SDL_HWSURFACE` (crea la superficie en la memora de vídeo), `SDL_DOUBLEBUF` (usamos doble *buffer*, sólo creando la superficie en memoria de vídeo), `SDL_FULLSCREEN` (la superficie será generada en pantalla completa).

***SDL_GetVideoSurface*():** devuelve el puntero a la superficie actual de *display*. Si no existe ninguna, devuelve un puntero nulo.

***SDL_Flip(*SDL_Surface *s)*:** intercambia el valor de los *buffers,* en el caso de que haya sido habilitada esa opción, devolviendo el valor del otro buffer.

***SDL_UpdateRect*(SDL_Surface *s, int x, int y, int w, int h):** actualiza la superficie indicada en la pantalla principal. Posicionandolo en la posición x, y con la altura y anchura indicadas.

***SDL_CreateRGBSurface*(Uint32 flags, int width, int height, int bitsPerPixel, Uint32 Rmask, Uint32 Gmask, Uint32 Bmask, Uint32 Amask):** crea una superficie vacía de tamaño y bits por píxel indicados. Los *flags pueden* ser los siguientes: `SDL_SWSURFACE` (crea la superficie en la memoria del sistema), `SDL_HWSURFACE` (crea la superficie en la memoria de video), `SDL_SRCCOLORKEY` (permite la existencia de colores transparentes en la superficie), `SDL_SRCALPHA` (permite la iluminación de colores en la superficie).

***SDL_CreateRGBSurfaceFrom*(void *pixels, Uint32 flags, int width, int height, int bitsPerPixel, Uint32 Rmask, Uint32 Gmask, Uint32 Bmask, Uint32 Amask):** crea una superficie de la misma forma que la anterior, pero a partir de la sucesión datos de los píxeles.

***SDL_FreeSurface*(SDL_Surface* surface):** libera los recursos y elimina una superficie de la memoria. Si la superficie está construida a partir de píxeles, los datos serán eliminados cuando se libere su zona de memoria.

***SDL_LockSurface*(SDL_Surface* surface):** bloquea el acceso directo a los píxeles de una superficie.

***SDL_UnlockSurface*(SDL_Surface* surface):** permite el acceso directo a los píxeles de una superficie previamente bloqueada.

***SDL_ConvertSurface*(SDL_Surface *src, SDL_PixelFormat *fmt, Uint32 flags):** devuelve una superficie copia de la dada por el parámetro *src*, aplicándole el formato y las opciones indicadas.

***SDL_DisplayFormat*(SDL_Surface *surface):** devuelve una superficie a partir de otra, aplicandole los colores y el formato de píxel del *buffer* de video, permitiendo un mejor pintado en el *display*.

***SDL_DisplayFormatAlpha*(SDL_Surface *surface):** igual que la anterior pero considerando también el factor de la iluminación.

***SDL_LoadBMP*(const char *file):** crea una superficie a partir de la imagen BMP con la dirección indicada por *file*.

***SDL_SaveBMP*(SDL_Surface *surface, const char *file):** graba el contenido de la superficie en formato BMP con el nombre de fichero indicado.

***SDL_SetClipRect*(SDL_Surface *surface, SDL_Rect *rect):** establece el rectángulo asociado a la superficie. Cuando la superficie se muestra, sólo se dibuja el área comprendida por el rectángulo.

***SDL_GetClipRect*(SDL_Surface *surface, SDL_Rect *rect):** devuelve el rectángulo asociado a al superficie indicada.

***SDL_BlitSurface*(SDL_Surface *src, SDL_Rect *srcrect, SDL_Surface *dst, SDL_Rect *dstrect):** copia el contenido de una superficie en otra indicada. Si se le pasa un SDL_Rect se copia el trozo indicado de la superficie origen en la zona indicada de la superficie destino.

***SDL_FillRect*(SDL_Surface *dst, SDL_Rect *dstrect, Uint32 color):** rellena una zona de una superficie de un color. Si *dstrect* es nulo, toda la superficie es rellenada del color indicado. El color de relleno es el generado por las funciones SDL_MapRGB y SDL_MapRGBA.

***SDL_MapRGB*(SDL_PixelFormat *fmt, Uint8 r, Uint8 g, Uint8 b):** obtiene la representación del color indicado por r,g,b según el formato de los píxeles.

***SDL_MapRGBA*(SDL_PixelFormat *fmt, Uint8 r, Uint8 g, Uint8 b, Uint8 a):** realiza lo mismo que la función anterior, pero incluyendo además la intensidad de iluminación.

***SDL_GetRGB*(Uint32 pixel, SDL_PixelFormat *fmt, Uint8 *r, Uint8 *g, Uint8 *b):** devuelve los valores r, g, b a partir de la representación del color de un píxel según el formato de éste.

***SDL_GetRGBA*(Uint32 pixel, SDL_PixelFormat *fmt, Uint8 *r, Uint8 *g, Uint8 *b, Uint8 *a):** realiza la misma función que la anterior, pero dando además el valor de iluminación.

***SDL_SetColors*(SDL_Surface *surface, SDL_Color *colors, int firstcolor, int ncolors):** actualiza los colores de la superficie a partir de un vector de colores indicando el primero y la cantidad de estos que se deben de tomar.

***SDL_SetPalette*(SDL_Surface *surface, int flags, SDL_Color *colors, int firstcolor, int ncolors):** establece la paleta de colores de la superficie a partir de un vector de colores, indicando el primero y la cantidad de estos.

***SDL_SetColorKey*(SDL_Surface *surface, Uint32 flag, Uint32 key):** establece para la superficie el color que será transparente. El color se indicar por el parámetro *key*, que viene dado por la representación de SDL_MAPRGB.

***SDL_SetAlpha*(SDL_Surface *surface, Uint32 flags, Uint8 alpha):** establece el nivel *alpha* de la superficie a partir del valor pasado por parámetros.

***rotozoomSurface*(SDL_Surface * src, double angle, double zoom, int smooth):** Devuelve la imagen *src* rotada tantos grados como los indicados por el parámetro *angle.* La imagen se puede escalar multiplicando por el valor de *zoom* y difuminarla con el valor *smooth*. La rotación se realiza tomando como centro punto medio de la imagen y en sentido anti horario. Los espacios nuevos que se generan debido a la rotación, ya que la imagen tiene que ser rectangular, se consideran píxeles negros y se almacenan de esa manera en la nueva imagen.

Eventos

Tipos:

SDL_Event: estructura que representa el tipo evento y que almacena el tipo de evento que ha ocurrido y las características de éstos.

Type: tipo de evento que ha sucedido.
Active: evento de activación (`SDL_ACTIVEEVENT`). Sucede cuando el ratón se posiciona dentro de la ventana.
Key: evento de teclado (`SDL_KEYBOARDEVENT`). Sucede cuando se pulsa una tecla de éste. Indica que tecla se ha pulsado y si se encuentra todavía pulsada o si se ha dejado de pulsar.
Motion: evento de movimiento de ratón (`SDL_MOUSEMOTION`). Sucede cuando el ratón se mueve, indicando la posición en la que se encuentra y la distancia movida.
Button: evento de click de ratón (`SDL_MOUSEBUTTONEVENT`). Sucede cada vez que se produce un click en la aplicación. El evento indica la posición de pulsación, cual de los botones ha sido pulsado y si se encuentra pulsado o ya se ha dejado de pulsar.
Resize: evento de cambio de tamaño de la ventana (`SDL_VIDEORESIZE`). Sucede cada vez que se produce un cambio en el tamaño de la ventana, indicando la nueva altura y anchura de ésta.
User: evento creado por el desarrollador del programa que puede ser capturado cuando es necesario y que sirve para enviar otro tipo de información diferente a la de los demás eventos.
Expose: evento de modificación de pantalla por una ocurrencia desde fuera de la aplicación (`SDL_VIDEOEXPOSE`).
Quit: evento que se produce cuando se produce una salida de la aplicación desarrollada (`SDL_QUIT`).

Funciones:

Durante el desarrollo de la aplicación se pueden suceder diferentes eventos que, si interesan al desarrollador, se deben poder capturar. Los eventos capturados se almacenan en una lista hasta que son analizados mediante las siguiente funciones:

***SDL_PollEvent*(SDL_Event *event):** esta función obtiene el primer evento almacenado de la lista (por orden de llegada), lo elimina de la lista y lo almacena en la variable *event* para que sea tratada.

***SDL_WaitEvent*(SDL_Event *event):** espera indefinidamente a la llegada de un evento, cuando este llega lo captura y lo almacena en la variable *event*. Si ocurre algún fallo durante la espera de eventos la función devuelve un 0.

***SDL_PushEvent*(SDL_Event *event):** introduce un evento en la lista para que sea leído. Esta función se usa para tratar eventos generados por el usuario y para realizar una comunicación entre varios procesos a través de la cola de la lista.

Anexo 5. Desarrollo temporal

En este apartado se analiza el desarrollo temporal del proyecto, indicando el comienzo de cada actividad realizada y su finalización. En la Figura A5.1 se incluye el diagrama de Gantt de la distribución temporal de las tareas realizadas.

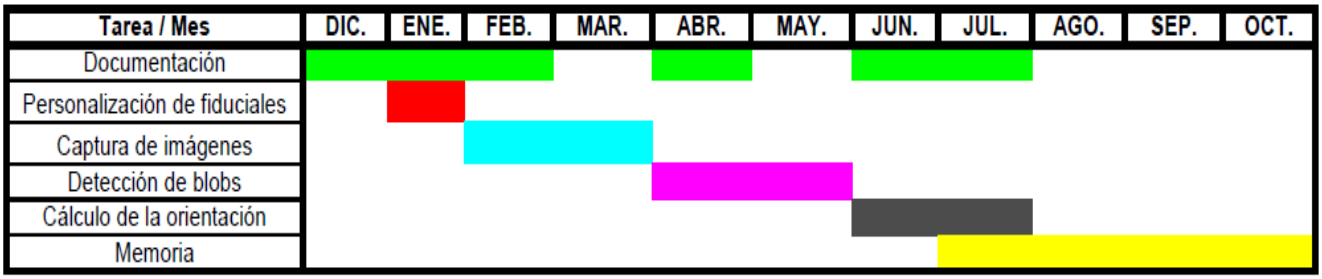

Figura A5.1 Diagrama de Gantt

La primera etapa de desarrollo del proyecto consistió en la documentación sobre el entorno de trabajo ReacTIVision. En esta etapa, que abarcó todo el mes de Diciembre y parte de los primeros días de Enero, se analizó todo el *framework* para comprender su funcionamiento, su comunicación, de qué elementos estaba compuesto, etc. Cuando se entendió la forma en la que trabajaba el *framework* se procedió al estudio de los fiduciales para entender sus características, la forma en que eran creados. A partir de algunos artículos [CHI09] se entendió la forma en que se constituían los *fiduciales* para poder construirlos sin depender de los que venían por defecto. En el mismo mes y gracias a las pruebas la parte de personalización quedó acabada y ya se detectaban los nuevos *fiduciales* en ReacTIVision.

En la siguiente etapa se procedió a realizar la captura de imágenes mediante el *tabletop*. Para ello se siguió analizando el *framework* para localizar el sitio exacto donde se realizaba la detección de los *fiduciales.* Además también se analizó la forma en que se usaba la librería *SDL* y sus funciones [VSDL], para utilizarla en el almacenamiento y tratamiento de las imágenes. La librería era usada por el *framework* con anterioridad para la carga de superficies y cuando se vio que permitía el almacenamiento y la carga de imágenes en formato BMP con facilidad, se optó por seguir utilizándola. A la vez que se iba analizando el código fuente del programa se procedía también a ir implementando las mejoras que se necesitaban, terminando de analizar el código y de entender el funcionamiento de la librería a finales de Febrero. El tiempo restante en Marzo se utilizó para mejorar la fase de captura de la imagen, tanto como las corrección de la orientación del folio, como el recorte y limpieza de la imagen final. A finales del mes se desarrolló la aplicación que permitía la captura de los dibujos para ser usada por los niños.

En Abril se procedió a implementar la funcionalidad de reconocimiento de blobs. Con la lectura de artículos nuevos que habían salido sobre el reconocimiento mejorado de *fiduciales* [GS09], [HaDt], [TuIts] se procedió a ver si servían para aportar información a lo que se necesitaba. Al ver que trataban el tema de mejorar los *fiduciales* y otros temas que no eran necesarios para lo que se estaba haciendo, se procedió a seguir analizando el código para detectar dónde se realizaba la detección de los *blobs* y cómo se hacía. A finales de Abril se comenzó a implementar la detección de *blobs*, siendo durante el mes

de Mayo cuando se procedió a realizar las pruebas con objetos diferentes y con las manos para ver si se detectaban bien, además de seguir mejorando la detección de estas con la parametrización y desarrollando el juego de golpear las imágenes que se muestran en el *tabletop*.

A finales de Mayo se introdujo la necesidad de intentar obtener la orientación de los objetos que se colocasen sobre la mesa para poder aplicarla a mejoras en los juegos. Mientras se seguía mejorando la detección de manos, se procedió a estudiar el tratamiento de imágenes y los momentos de éstas [MV]. Durante el mes de Junio se estuvo tratando el tema de la orientación, siendo a principios de Julio cuando la orientación se calculaba correctamente. Durante el tiempo que se estuvo implementando la forma de calcular la orientación también se analizó la estructura de comunicación TUIO para conseguir enviar la nueva información calculada de los *blobs* mediante el protocolo sin alterar su estructura mucho. Al principio se optó por intentar enviar la información de la orientación como si de un *fiducial* se tratase para aprovechar su campo de orientación, pero viendo que eso provocaba problemas en la comunicación, se optó por modificar el formato de envío de los *blobs.*

Una vez comprobado durante el mes de Julio que todas las mejoras implementadas funcionaban y que todos los objetivos se habían cumplido, se procedió a la redacción de la memoria y a la captura de imágenes y vídeos del funcionamiento real de las aplicaciones desarrolladas, a finales de este mes. Durante los meses de Agosto, Septiembre y parte de Octubre se ha procedido a realización de la memoria siendo ésta finalizada en este último mes y dando el proyecto como completado.

Anexo 6. Otros Tabletops existentes

En este anexo se analizarán algunos de los proyectos sobre *tabletops* existentes más relevantes que existen en el mundo, comentado sus características y comparando sus funcionalidades con respecto a las del *tabletop* usado.

6.1 Zach Lieberman – Drawn

Este *tabletop* permite al usuario dibujar sobre un folio con tinta para ser capturado y poder interactuar con las partes de este. Para realizar la captura del folio es necesario pulsar un botón incluido en la estructura de la mesa. Cuando el dibujo es capturado, el programa separa los elementos independientes del dibujo y permite la interacción del usuario con ellos [ZL].

En la Figura A6.1 se observan el proceso de dibujo y de manipulación del dibujo.

Figura A6.1 *Tabletop* Zach Lieberman – Drawn. A la izquierda se observa el proceso de dibujo. En la imagen de la derecha se observa la interacción con las partes del dibujo una vez capturado.

Existen varias diferencias con respecto a lo implementado en el proyecto. En el caso del proyecto realizado al incluir un *fiducial* al folio y permitir que la imagen sea capturada sin tener que pulsar ningún botón permite que sea más automático que de la otra forma. Como la necesidad era la de capturar el dibujo más que interaccionar con sus elementos se prosiguió con la implementación, si bien, se tomó la idea de incluir la interacción con los elementos de los dibujos una vez que el proceso de captura se finalizase. También existe la diferencia en el proyecto de que durante la interacción con el dibujo se pueden incluir más dibujos, mientras que en el desarrollado por Zach Lieberman no se puede.

6.2 Colaborative Tabletop

Tabletop colaborativo que permite la interacción de varios niños de edades superiores a los 6 años mediante un juego de construcción de un hilo de una historia colaborando entre ellos para construirla. El *tabletop* presenta una zona intermedia donde se depositan diapositivas para que el niño interactúe con ellas, arrastrándolas hacia su zona de juego mediante las manos o mediante el uso del ratón. Este *tabletop* permite la interacción mediante el uso de manos con las imágenes que se imprimen sobre la superficie al igual que lo desarrollado en el proyecto además de incluir la opción de poder usar el ratón. La opción de incluir ratón en el proyecto que se ha realizado no se

contempló ya que lo que se pretende es que los niños usen los elementos de juego con los que están más familiarizados y que estos niños sean de edades inferiores a los 6 años [IRI]. En la Figura A6.2 se muestra la interacción de los niños con el *tabletop* y la forma de éste.

Figura A6.2 Interface del Colaborative Tabletop. Muestra el centro del *tabletop* en el que se dibujan las diapositivas para ordenarlas luego en los extremos.

6.3 Flux Digital Tabletop

Tabletop de tratamiento gráfico que permite el dibujo sobre la superficie mediante el uso de lápices de diferentes tamaños de trazo. Diseñado para realizar dibujos precisos permite también la interacción de varias personas con él, además de incluir la posibilidad de ampliar las zonas de la imagen. Cuando se posiciona la punta de uno de los lápices sobre la superficie del *tabletop* este la reconoce y va dibujando, siguiendo el recorrido que se haga, un línea. Usando el movimiento con los dedos se puede hacer zoom a las diferentes partes del dibujo para añadir detalles con más precisión, o guardar y cargar otros dibujos diferentes realizados con anterioridad [Flux].

Este *tabletop* está diseñado para el dibujo preciso en la realización de planos de estructuras, como se puede observar en la Figura A6.3. En el *tabletop* usado durante el proyecto no se permite el dibujo sobre la superficie, ya que para el niño es más normal dibujar sobre un papel de folio normal. La opción de ampliar las zonas de la imagen tampoco son necesarias ya que lo importante es que el niño pueda interaccionar con lo dibujado y mostrado y no que realice un dibujo preciso. En la Figura A6.4 se puede observar el movimiento de las manos para realizar el zoom al dibujo, haciendo el movimiento de dentro hacia fuera para aumentarlo, o al revés, para disminuirlo.

Figura A6.3 Flux Digital Tabletop. En la imagen se observa la superficie de la mesa y como se va dibujando precisamente sobre ella.

Figura A6.4 Imagen de la acción para aumentar el zoom del dibujo que se está tratando.

6.4 U-Touch 103'' Multitouch Air Hockey

Tabletop que simula el juego de una mesa de Hockey Aéreo. La mesa detecta la posición del dedo de los usuarios, dibujando el mando centrado en el punto de interacción siguiendo la trayectoria de este cuando se desplaza y calculando la dirección que tomará el disco cuando se le golpee. La detección se realiza mediante la detección de *fingers* que ya viene implementada en el propio programa de ReacTIVision, realizando un cálculo de trayectoria y de rebotes para mostrar el movimiento del disco como si fuese real, aparte de permitir la interacción de más de un usuario [UTAH]. En la Figura A6.4 se muestra una imagen de la aplicación siendo usada por 2 personas.

En referencia al trabajo desarrollado en el proyecto se observa que se realiza interacción con elementos digitales además de detectar el movimiento que realizan los jugadores con el dedo. En este caso el dedo interacciona con el disco, mientras que en el que se ha desarrollado el proyecto se interacción con los dibujos capturados.

Figura A6.4 Mesa de juego U-Touch 103" Multitouch Air Hockey

6.5 Reactable

Este *tabletop* permite la creación musical mediante un ordenador. Sus desarrolladores implementaron el *framework* de ReacTIVision para poder detectar la colocación de diferentes elementos sobre la superficie del *tabletop*. Mediante la colocación de cubos con *fiduciales* aplicados bajo una de sus caras se generan los elementos sonoros y a través de la orientación de éstos y la distancia entre los elementos se puede elegir la frecuencia con la que son generados los sonidos. Una vez que la aplicación recibe los valores de la posición de los elementos y su orientación se encarga de generar el audio respecto a estos valores [RT]. Se ha considerado que incluir una descripción de este *tabletop* era necesaria, ya que el *framework* usado durante el proyecto es el que se desarrolló para realizar Reactable. En la Figura A6.5 se muestra el funcionamiento y la variación de frecuencia mediante la orientación y distancia.

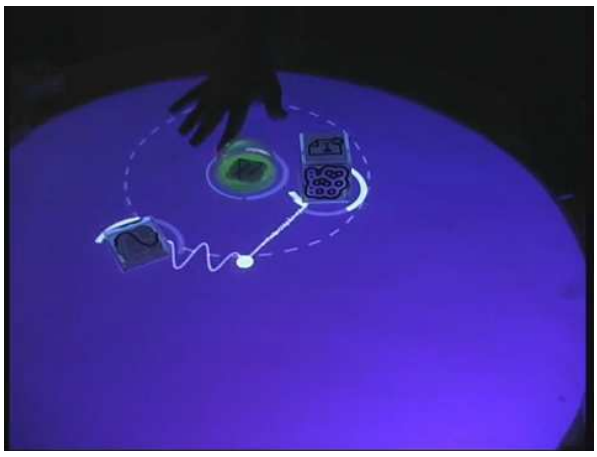
Figura A6.5 Parte superior del *tabletop* Reactable. En la imagen se puede observar que dependiendo de la distancia donde se coloquen los elementos, la posición y que *fiducial* se generan diferentes sonidos y formas de onda.

Bibliografía

[ChiCI] Child Computer Interaction Group
http://www.chici.org/janet_read.php

[FID] Flexible fiducial generator for reacTIVision based projects
http://code.google.com/p/fidgen/

[REAC] ReacTIVision
http://reactivision.sourceforge.net/

[CH09] Designable Visual Markers
Enrico Costanza, Jeffrey Huang

[GS09] Recognition, Tracking and Association of Hands, Fingers, and Blobs: A Tbeta Upgrade.
hiago de Freitas Oliveira Araújo

[HaDt] Detecting Hands, Fingers and Blobs for Multi-Touch Display Applications
Thiago de Freitas Oliveira Araújo, Alexsandro J. V. dos Santos.

[TuIts] reacTIVision and TUIO: A Tangible Tabletop Toolkit
Martin Kaltenbrunner

[DIMG] Descriptores de Imagen
Marcos Martín
http://poseidon.tel.uva.es/~carlos/ltif10001/descriptores.pdf

[MV] Machine Vision
Ramesh Jain, Rangachar Kasturi, Brian G. Schunck
McGraw-Hill, Inc., ISBN 0-07-032018-7, 1995
http://www.cse.usf.edu/~r1k/MachineVisionBook/MachineVision.pdf

[ZL] Thesystemis. The works of Zach Lieberman & Collaborators
http://thesystemis.com/

[RT] Reactable
http://www.reactable.com

[TUIO] TUIO
http://www.tuio.org/

[VSDL] Programación de Videojuegos con SDL.
Alberto García Serrano. 2003 Ediversitas Multimedia SL

[IRI] Informatics Research Institute
http://irgen.ncl.ac.uk/wiki/bin/view/Undergrad/RichardWhite

[Flux] Flux - Fully Liberating User eXperience
http://mi-lab.org/projects/flux/

[UTAH] U-Touch 103" Multitouch Air Hockey
http://wn.com/U-Touch_103_Multitouch_Air_Hockey

[HEBJ] Getting a grip on tangible interaction: a framework on physical space and social interaction. *Hornecker E., Buur J. (2006)*
Proc. Of CHI'06, ACM Press.